SQL BASICS

Structured Query Language

The Language of Databases

- Easy to Learn
- Powerful
- Industry Standard
- High Career Opportunities

1 SQL LOGO

SQL stands for **Structured Query Language.**
It is used to communicate with databases.

2 DATABASE

A **Database** is a collection of related data stored in an organized way.

3 TABLE

Table: students

id	name	age	course
1	Aman	20	BCA
2	Simran	21	MCA
3	Karan	22	B.Tech
4	Priya	20	BCA

A **Table** is a structure in a database that stores data in rows and columns.

4 COLUMNS

Column (Field)

id	name	age	course
1	Aman	20	BCA
2	Simran	21	MCA
3	Karan	22	B.Tech
4	Priya	20	BCA

Columns are the fields of a table. Each column has a specific type of data.

5 ROWS

Row (Tuple) →

id	name	age	course
1	Aman	20	BCA
2	Simran	21	MCA
3	Karan	22	B.Tech
4	Priya	20	BCA

Rows are individual entries in a table.
Each row is a single record.

6 RECORDS

ID: 2
Name: Simran
Age: 21
Course: MCA

A **Record** = All the column values of a single row.

7 QUERIES

SQL Queries are used to fetch, insert, update, delete and manage data.

Common Queries:

- SELECT – Read data
- INSERT – Add data
- UPDATE – Modify data
- DELETE – Remove data
- CREATE – Create table
- DROP – Delete table

SQL EXAMPLES

```sql
-- Create a table
CREATE TABLE students (
  id INT PRIMARY KEY,
  name VARCHAR(50),
  age INT,
  course VARCHAR(50)
);
```

```sql
-- Insert data
INSERT INTO students (id, name, age, course)
VALUES (1, 'Aman', 20, 'BCA');
```

```sql
-- Read data
SELECT * FROM students;
```

```sql
-- Update data
UPDATE students
SET age = 21
WHERE name = 'Aman';
```

```sql
-- Delete data
DELETE FROM students
WHERE id = 1;
```

SQL Workflow

1. Write SQL Query
2. Database Engine
3. Process Query
4. Return Result

Popular Databases

- MySQL
- PostgreSQL
- SQLite
- Microsoft SQL Server
- ORACLE
- MariaDB

Why Learn SQL?

- Essential for Web Development
- Used in Apps & Software
- High Demand Skills
- Works with any Programming Language
- Great for Data Analysis

Real-World Uses

Web Apps · Banking · E-Commerce · Healthcare · Education · Analytics

Key Takeaways

- SQL = Structured Query Language
- Data is stored in Databases → Tables → Rows → Columns
- Records are complete rows
- Queries help us manage data easily

"Learn SQL Today, Build Better Apps Tomorrow!"

SQL INSTALLATION

Popular SQL Databases – Installation, Workbench & Connection

- **Easy to Install** – Quick setup
- **Open Source** – Free to use
- **Powerful** – High Performance
- **Secure** – Enterprise Ready

WHAT IS SQL?

SQL (Structured Query Language) is used to create, manage, and manipulate relational databases.

DATABASE

A database is a collection of related data.
Managed using SQL.

WORKBENCH

A tool or interface used to manage databases, run queries and visualize data.

CONNECTION

A connection allows you to connect your workbench to the database server.

1 MySQL

MySQL

Installation Steps

1. Download from mysql.com
2. Run Installer
3. Select Server, Workbench and other tools
4. Configure and install
5. Set root password
6. Finish

Workbench / Tool

MySQL Workbench

Official Website: https://www.mysql.com

2 PostgreSQL

PostgreSQL

Installation Steps

1. Download from postgresql.org
2. Run Installer
3. Select components (Server, pgAdmin, etc.)
4. Set password for user
5. Complete installation

Workbench / Tool

pgAdmin 4

Official Website: https://www.postgresql.org

3 SQLite

SQLite

Installation Steps

1. Download from sqlite.org
2. Extract the package
3. No installation required
4. Use SQLite files directly

Workbench / Tool

DB Browser for SQLite

Official Website: https://www.sqlite.org

4 SQL Server

Microsoft SQL Server

Installation Steps

1. Download from microsoft.com
2. Run Installation Center
3. Select SQL Server features
4. Configure instance
5. Complete installation

Workbench / Tool

SQL Server Management Studio (SSMS)

Official Website: https://www.microsoft.com/sql-server

5 Oracle Database

ORACLE DATABASE

Installation Steps

1. Download from oracle.com
2. Unzip the package
3. Run runInstaller
4. Select database edition
5. Configure and install
6. Finish

Workbench / Tool

SQL Developer

Official Website: https://www.oracle.com

POPULAR TOOLS / WORKBENCHES

- MySQL Workbench
- pgAdmin 4
- DB Browser for SQLite
- SSMS
- Oracle SQL Developer

HOW TO CONNECT (Example)

MySQL Workbench
Host: localhost
Port: 3306
User: root
Password: ******
Connect

pgAdmin 4
Host: localhost
Port: 5432
User: postgres
Password: ******
Connect

DB Browser for SQLite
File: database.sqlite
(Open Database)
Connect

SSMS
Server: localhost\SQLEXPRESS
User: sa
Password: ******
Connect

SQL Developer
Host: localhost
Port: 1521
Service: XE
User: system
Password: ******
Connect

TIPS

- Use strong passwords
- Keep your database software updated
- Open required ports in firewall
- Regularly backup your databases

LEARN SQL

After installation, start learning SQL queries to create, read, update and manage data efficiently.

DATABASE BASICS

Building Blocks of a Database

WHAT IS A DATABASE?

A database is an organized collection of related data that can be easily accessed, managed and updated.

DATABASE

A database is a collection of related data organized to be easily accessed, managed and updated.

Example: SchoolDB

TABLE

A table stores data in rows and columns. It is like a spreadsheet in a database.

Example: Students

SCHEMA

A schema is the blueprint of the database. It defines tables, columns, data types, and relationships.

Example: SchoolDB Schema

TABLE EXAMPLE: Students

Columns

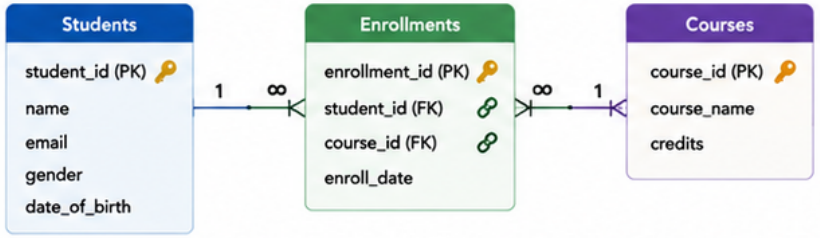
Primary Key
Uniquely identifies each row.

student_id (INT)	name (VARCHAR)	email (VARCHAR)	gender (CHAR)	date_of_birth (DATE)
1	Alice	alice@email.com	F	2004-01-15
2	Bob	bob@email.com	M	2004-03-22
3	Charlie	charlie@email.com	M	2004-05-30
4	Diana	diana@email.com	F	2004-07-18
5	Eve	eve@email.com	F	2004-09-09

ROW (RECORD)
A row is a single entry in a table.

COLUMNS
A column represents an attribute (field) of the table.

RELATIONSHIP BETWEEN TABLES



Relationships connect tables using keys. They help maintain data integrity and reduce redundancy.

PK = Primary Key
FK = Foreign Key
1 = One
∞ = Many

ROW (RECORD)

A row (record) is a single entry in a table. It contains related information.

Example: Alice's information

COLUMN

A column is a field that stores a specific type of data.

Example: name, email

PRIMARY KEY

A primary key uniquely identifies each row in a table.

Example: student_id

QUICK SUMMARY

- Database stores related data.
- Tables organize data in rows and columns.
- Each row is a record.
- Each column is an attribute.
- Primary key uniquely identifies records.
- Schema defines the structure.
- Relationships connect tables.

TIP: Good database design makes your data consistent and easy to manage.

BEST PRACTICES

- ✓ Use meaningful table and column names
- ✓ Define primary keys for every table
- ✓ Avoid duplicate data (use relationships)
- ✓ Use proper data types

CREATE DATABASE

SQL Commands to Manage Databases

A database is a container for tables, views, procedures and other database objects.
Use SQL commands to create, view, use and delete databases.

POPULAR SQL PLATFORMS

MySQL

SQL Server

1 CREATE DATABASE

Creates a new database in the system.

MySQL

```
CREATE DATABASE mydb;
```

SQL Server

```
CREATE DATABASE MyDB;
```

2 USE DATABASE

Selects a database to work with.

MySQL

```
USE mydb;
```

SQL Server

```
USE MyDB;
```

3 DROP DATABASE

Deletes an existing database.

MySQL

```
DROP DATABASE mydb;
```

SQL Server

```
DROP DATABASE MyDB;
```

4 SHOW DATABASES

Displays all databases available in the server.

MySQL

```
SHOW DATABASES;
```

SQL Server

```
SHOW DATABASES;
```

5 DATABASE SCHEMA

Defines the structure of the database (tables, types, etc.).

MySQL

```
DESCRIBE mydb;
```

SQL Server

```
EXEC sp_helpdb MyDB;
```

WHAT IS A SCHEMA?

A schema is the logical structure of a database. It includes tables, views, indexes, relationships, and other objects.

Example: mydb schema contains all its tables and objects.

STORAGE

Databases are stored on disk. They consume storage space based on data and objects.

Example: mydb stores data, indexes, logs, and more.

EXAMPLE WORKFLOW (MySQL)

1. Create — `CREATE DATABASE mydb;`
2. Use — `USE mydb;`
3. Show — `SHOW DATABASES;`
4. Work with — Create tables, insert data, etc.
5. Drop (if needed) — `DROP DATABASE mydb;`

HOW DATABASE IS STORED?

Database: mydb

Storage Components:

- Tables (Data)
- Indexes (Fast Search)
- Logs (Transactions)
- Other Objects (Views, Proc, etc.)

KEY POINTS

- Database is a top-level container.
- Each database has its own schema.
- Use USE to select the database.
- Use DROP carefully (data will be lost).
- SHOW DATABASES lists all databases.

QUICK COMMAND REFERENCE

Action	MySQL	SQL Server	Description
Create Database	`CREATE DATABASE mydb;`	`CREATE DATABASE MyDB;`	Creates a new database.
Use Database	`USE mydb;`	`USE MyDB;`	Selects the database to work with.
Show Databases	`SHOW DATABASES;`	`SHOW DATABASES;`	Shows all databases.
Drop Database	`DROP DATABASE mydb;`	`DROP DATABASE MyDB;`	Deletes an existing database.
Show Schema Info	`DESCRIBE mydb;`	`EXEC sp_helpdb MyDB;`	Shows schema / structure information.

TIP

Always backup your database before dropping it.
Be careful while using DROP DATABASE command.

CREATE TABLE

Define Structure. Store Data.

The **CREATE TABLE** statement is used to create a new table in a database.

Works with MySQL and Microsoft SQL Server

KEY POINTS

- A table stores data in rows and columns.
- Design your table before inserting data.
- Choose appropriate data types.
- Use **constraints** to maintain data integrity.
- **PRIMARY KEY** uniquely identifies each row.
- AUTO_INCREMENT generates unique IDs.

1 CREATE TABLE SYNTAX

MySQL Syntax

```
CREATE TABLE table_name (
    column1 data_type constraint,
    column2 data_type constraint,
    column3 data_type constraint,
    ...
    table_constraints
);
```

SQL Server Syntax

```
CREATE TABLE table_name (
    column1 data_type constraint,
    column2 data_type constraint,
    column3 data_type constraint,
    ...
    table_constraints
);
```

2 COLUMNS

Columns are the fields of a table.

Column Name	Description
id	Unique identifier for each row
name	Name of the person
email	Email address
age	Age in years
created_at	Record creation date & time

3 DATA TYPES (MySQL)

Commonly used data types.

Data Type	Description
INT	Integer numbers
VARCHAR(n)	Variable length string (n = max length)
CHAR(n)	Fixed length string
DATE	Date (YYYY-MM-DD)
DATETIME	Date and time
BOOLEAN	True / False (0 / 1)
DECIMAL(p,s)	Decimal numbers (p = precision, s = scale)
TEXT	Long text data

4 CONSTRAINTS

Rules applied to columns.

PRIMARY KEY
Uniquely identifies each row.

NOT NULL
Column cannot have NULL values.

UNIQUE
All values must be unique.

DEFAULT
Sets a default value.

CHECK
Ensures values satisfy a condition.

FOREIGN KEY
Links to another table's key.

5 PRIMARY KEY

A PRIMARY KEY uniquely identifies each row.

id	name	email
1	Alice	alice@example.com
2	Bob	bob@example.com
3	Charlie	charlie@example.com

6 AUTO_INCREMENT

Automatically generates unique numbers.

id (AUTO_INCREMENT)	name
1	Alice
2	Bob
3	Charlie
4	Diana

7 SCHEMA

Schema is the blueprint (structure) of the database.

8 SAMPLE TABLE DESIGN

Example: users table

```
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    age INT CHECK (age >= 0),
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

users

Field	Type	Constraints	Key	Extra
id	INT	NOT NULL	PRIMARY KEY	AUTO_INCREMENT
name	VARCHAR(100)	NOT NULL	-	-
email	VARCHAR(100)	NOT NULL, UNIQUE	-	-
age	INT	CHECK (age >= 0)	-	-
created_at	DATETIME	DEFAULT CURRENT_TIMESTAMP	-	-

9 TABLE DESIGN TIPS

- Use meaningful table and column names.
- Choose correct data types.
- Always define a PRIMARY KEY.
- Use constraints to avoid invalid data.
- Plan relationships between tables.

QUICK EXAMPLES (MySQL)

Simple Table

```
CREATE TABLE products (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    price DECIMAL(10,2) NOT NULL
);
```

Table with Constraints

```
CREATE TABLE employees (
    emp_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    salary DECIMAL(10,2) CHECK (salary >= 0)
);
```

Table with Default

```
CREATE TABLE posts (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(200) NOT NULL,
    status VARCHAR(20) DEFAULT 'draft',
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

Table with Foreign Key

```
CREATE TABLE orders (
    order_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    amount DECIMAL(10,2) NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

TIP

Good table design = Strong Database
Plan well, code less. Think before you create!

SQL DATA TYPES

Choose the right data type to store the right kind of data.

Data types define the kind of data that can be stored in a column. Using correct data types improves performance and saves storage.

DATA TYPE	DESCRIPTION	SYNTAX (EXAMPLE)	EXAMPLE VALUE	STORAGE	RANGE / SIZE
INT 123	Stores whole numbers (integers).	`INT` `INT(11)`	101, -25, 0, 9999	4 Bytes	-2,147,483,648 to 2,147,483,647
VARCHAR abc	Variable-length string. Maximum length is defined.	`VARCHAR(n)` `VARCHAR(50)`	"John Doe", "Hello World"	n + 1 Bytes	0 to 65,535 characters
CHAR ab	Fixed-length string. Spaces are padded.	`CHAR(n)` `CHAR(10)`	"A", "ABC"	n Bytes	0 to 255 characters
TEXT	Stores large text data.	`TEXT`	"This is a long text with many words..."	Up to 64 KB (MySQL)	0 to 65,535 bytes
DATE	Stores date values.	`DATE`	"2025-05-26"	3 Bytes	'1000-01-01' to '9999-12-31'
TIME	Stores time values.	`TIME`	"14:30:45"	3 Bytes	'-838:59:59' to '838:59:59'
BOOLEAN	Stores true or false values.	`BOOLEAN`	`TRUE`, `FALSE` (1, 0)	1 Byte	TRUE / FALSE or 1 / 0
FLOAT 3.14	Stores floating-point (approximate) numbers.	`FLOAT(p, s)` `FLOAT(10,2)`	3.14, -0.01, 99.99	4 Bytes	Approximate values (Precision based)

QUICK REFERENCE

Data Type	Use For
INT	Whole numbers
VARCHAR	Variable-length text
CHAR	Fixed-length text
TEXT	Large text data
DATE	Date only
TIME	Time only
BOOLEAN	True / False values
FLOAT	Decimal / real numbers

KEY POINTS

- Use INT for numbers without decimals.
- Use VARCHAR for text with varying length.
- Use CHAR for fixed-length codes (e.g., ISO codes).
- Use TEXT for large descriptions or documents.
- Use DATE and TIME for date and time values.
- Use BOOLEAN for flags or yes/no values.
- Use FLOAT/DECIMAL for decimal numbers.

EXAMPLE TABLE

```
CREATE TABLE employees (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    code CHAR(10),
    bio TEXT,
    birth_date DATE,
    join_time TIME,
    is_active BOOLEAN,
    salary FLOAT(10,2)
);
```

SAMPLE DATA

id	name	code	bio	birth_date	join_time	is_active	salary
1	Alice	EMP001	Team Lead	1995-04-12	09:30:00	TRUE	55000.50
2	Bob	EMP002	Developer	1996-07-20	10:15:30	TRUE	42000.00
3	Charlie	EMP003	Intern	2001-11-05	09:00:00	FALSE	15000.75

STORAGE TIP

Choosing the right data type reduces storage usage and improves query performance.

BEST PRACTICES

- Always use the smallest appropriate data type.
- Avoid using TEXT for short strings.
- Use BOOLEAN for flags instead of TINYINT(1).
- Correct data types lead to better performance.

INSERT DATA

Add new records (rows) into a table.

TABLE — A collection of related data in rows and columns.

RECORD — A single row in a table.

DATA ENTRY — Inserting data adds new records to the table.

BASIC SYNTAX

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

1 INSERT INTO & VALUES

Insert a single row into a table.

```
INSERT INTO employees (id, name, position, salary)
VALUES (1, 'Alice', 'Developer', 55000);
```

Explanation:
Inserts one record with values for the specified columns.

2 INSERT MULTIPLE ROWS

Insert multiple rows in a single query.

```
INSERT INTO employees (id, name, position, salary)
VALUES (2, 'Bob', 'Designer', 48000),
       (3, 'Charlie', 'Tester', 42000),
       (4, 'Diana', 'HR', 39000);
```

Explanation:
Faster way to insert multiple records at once.

COMMON COLUMN BEHAVIOR

Column Type/Setting	Behavior
NOT NULL	Must provide a value.
NULL	Can store NULL if allowed.
DEFAULT value	Used if no value is provided.
AUTO_INCREMENT	Generates next number automatically.

EXAMPLE TABLE: employees

id (INT)	name (VARCHAR)	position (VARCHAR)	salary (DECIMAL)	join_date (DATE)	is_active (BOOLEAN)
1	Alice	Developer	55000.00	2024-01-10	TRUE
2	Bob	Designer	48000.00	2024-01-12	TRUE
3	Charlie	Tester	42000.00	2024-01-15	TRUE
4	Diana	HR	39000.00	2024-01-18	TRUE

Each row represents a record. Using INSERT, we add new records to this table.

HOW INSERT WORKS

1. Table — Choose the table where data will be inserted.

2. Columns — Specify the columns (optional). If not specified, all columns must be provided.

3. Values — Provide values in the same order as the columns.

4. New Record — A new record (row) is added to the table.

PRACTICAL EXAMPLE (MySQL)

```
-- Single row
INSERT INTO employees (id, name, position, salary, join_date, is_active)
VALUES (7, 'Grace', 'Manager', 75000, '2024-01-20', TRUE);

-- Multiple rows
INSERT INTO employees (id, name, position, salary, join_date, is_active)
VALUES (8, 'Henry', 'Developer', 60000, '2024-01-22', TRUE),
       (9, 'Ivy', 'Tester', 45000, '2024-01-323', TRUE);
-- With NULL
INSERT INTO employees (id, name, position, salary, join_date, is_active)
VALUES (10, 'Jack', 'Analyst', NULL, '2024-01-25', FALSE);
-- Using DEFAULT (salary DEFAULT 30000, is_active DEFAULT TRUE)
INSERT INTO employees (id, name, position)  VALUES (11, 'Kate', 'Support');
```

3 USING NULL

Insert NULL when a value is unknown or not applicable.

```
INSERT INTO employees (id, name, position, salary, join_date)
VALUES (5, 'Eve', 'Intern', 15000, NULL);
```

NULL means "no value" or "unknown value".

4 DEFAULT VALUE

If a column has a DEFAULT value, you can skip it.

```
-- salary has DEFAULT 30000, is_active has DEFAULT TRUE
INSERT INTO employees (id, name, position)
VALUES (6, 'Frank', 'Support');
```

Columns with DEFAULT will automatically use their default values.

QUICK SUMMARY

Concept	Description
INSERT INTO	Adds new records to a table.
VALUES	Specifies the values to be inserted.
Multiple Rows	Insert many records in one query.
NULL	Represents missing or unknown value.
DEFAULT	Automatically uses the column's default value.
Record	A single row of data in a table.

BEST PRACTICES

- Always specify column names for better readability.
- Make sure the number of values matches the columns.
- Use appropriate data types for each column.
- Validate data before inserting to avoid errors.

SELECT DATA

Retrieve data from a table using SELECT statement.

HOW SELECT WORKS

QUERY — Write SELECT statement

DATABASE — Query is executed on the database

RESULT SET — Matching rows are returned

YOU SEE DATA — Data is shown in the result

BASIC SYNTAX

```
SELECT column1, column2, ....
FROM table_name
WHERE condition;

-- Get all columns
SELECT * FROM table_name;
```

1 SELECT ALL COLUMNS (*)

Use * to select all columns from a table.

```
SELECT * FROM employees;
```

Result

id	name	position	salary	join_date	is_active
1	Alice	Developer	55000.00	2024-01-10	1
2	Bob	Designer	48000.00	2024-01-12	1
3	Charlie	Tester	42000.00	2024-01-15	1
4	Diana	HR	39000.00	2024-01-18	1

2 SELECT SPECIFIC COLUMNS

Select only the columns you need.

```
SELECT id, name, salary FROM employees;
```

Result

id	name	salary
1	Alice	55000.00
2	Bob	48000.00
3	Charlie	42000.00
4	Diana	39000.00

3 COLUMN NAME & ALIAS

Use column names and aliases (AS) for better readability.

```
SELECT name AS employee_name, salary AS pay
FROM employees;
```

Result

employee_name	pay
Alice	55000.00
Bob	48000.00
Charlie	42000.00
Diana	39000.00

KEY TERMS

SELECT	Keyword used to retrieve data.
FROM	Specifies the table to retrieve data from.
*	Represents all columns in the table.
Column Name	Column(s) you want to retrieve.
Alias	Temporary name for a column or table.
Result	Output returned by the query.
Rows	Each row is a record in the result.
Query	SQL command to get data.

4 FILTER ROWS WITH WHERE

Retrieve only rows that match a condition.

```
SELECT id, name, position, salary
FROM employees
WHERE salary > 45000;
```

Result

id	name	position	salary
1	Alice	Developer	55000.00
2	Bob	Designer	48000.00

5 ORDER RESULTS

Sort the result using ORDER BY.

```
SELECT name, salary
FROM employees
ORDER BY salary DESC;
```

Result

name	salary
Alice	55000.00
Bob	48000.00
Charlie	42000.00
Diana	39000.00

6 LIMIT ROWS

Limit the number of rows returned.

```
SELECT * FROM employees
ORDER BY id
LIMIT 2;
```

Result

id	name	position	salary	join_date	is_active
1	Alice	Developer	55000.00	2024-01-10	1
2	Bob	Designer	48000.00	2024-01-12	1

SAMPLE QUERY BREAKDOWN

```
SELECT name AS employee_name, salary   → Columns with alias
FROM employees                         → Table name
WHERE is_active = 1                    → Filter condition
ORDER BY salary DESC                   → Sort order
LIMIT 3;                               → Limit rows
```

COMMON SELECT EXAMPLES

Description	Query	What it does
Select all columns	`SELECT * FROM employees;`	Returns all columns and all rows.
Select specific columns	`SELECT id, name, salary FROM employees;`	Returns only id, name and salary columns.
Use alias	`SELECT name AS emp, salary AS pay FROM employees;`	Uses aliases for column names.
Filter rows	`SELECT * FROM employees WHERE is_active = 1;`	Returns only active employees.
Sort results	`SELECT * FROM employees ORDER BY salary DESC;`	Returns rows sorted by salary (high to low).
Limit results	`SELECT * FROM employees LIMIT 3;`	Returns only first 3 rows.

TABLE: employees

Column	Data Type	Description
id	INT	Employee ID (Primary Key)
name	VARCHAR(100)	Employee name
position	VARCHAR(100)	Job position
salary	DECIMAL(10,2)	Salary amount
join_date	DATE	Date of joining
is_active	BOOLEAN	Active (1) or Inactive (0)

QUICK TIPS

- Use * for quick results, but specify columns in production.
- Use meaningful aliases for better readability.
- Always use WHERE to filter unnecessary data.
- ORDER BY helps to sort results.
- LIMIT is useful for previewing or pagination.

BEST PRACTICE — Always select only the columns you need. It improves performance.

TIP — Test your queries with LIMIT before running on large tables.

NOTE — Large result sets can slow down your application.

WHERE CLAUSE

Filter data by setting conditions.

How WHERE Clause Works

WHAT IS WHERE CLAUSE?

The WHERE clause is used to filter records that meet a specific condition. It is used with SELECT, UPDATE, DELETE statements.

COMPARISON OPERATORS

Operator	Meaning	Example	Result
=	Equal to	salary = 50000	Equal values
!=	Not equal to	salary != 50000	Not equal values
>	Greater than	salary > 50000	Greater values
<	Less than	salary < 50000	Lesser values
>=	Greater than or equal to	salary >= 50000	Greater or equal values
<=	Less than or equal to	salary <= 50000	Lesser or equal values

SAMPLE TABLE: employees

id	name	position	salary	department	join_date
1	Alice	Developer	55000	IT	2024-01-10
2	Bob	Designer	45000	Design	2024-01-12
3	Charlie	Tester	38000	QA	2024-01-15
4	Diana	HR	42000	HR	2024-01-18
5	Eve	Developer	60000	IT	2024-01-20
6	Frank	Manager	75000	IT	2024-01-22
7	Grace	Intern	25000	HR	2024-01-25

EXAMPLES WITH RESULTS

Condition	Query	Result (Matching Rows)
salary = 50000	`SELECT * FROM employees WHERE salary = 50000;`	No rows
salary != 50000	`SELECT * FROM employees WHERE salary != 50000;`	All except 50000
salary > 50000	`SELECT * FROM employees WHERE salary > 50000;`	Alice, Eve, Frank
salary < 50000	`SELECT * FROM employees WHERE salary < 50000;`	Bob, Charlie, Diana, Grace
salary >= 50000	`SELECT * FROM employees WHERE salary >= 50000;`	Alice, Eve, Frank
salary <= 50000	`SELECT * FROM employees WHERE salary <= 50000;`	Bob, Charlie, Diana, Grace

FILTER DATA EXAMPLES

Using comparison operators in WHERE clause.

1. `SELECT * FROM employees WHERE salary = 50000;`
   `-- Employees with salary equal to 50000`
2. `SELECT * FROM employees WHERE salary != 50000;`
   `-- Employees with salary not equal to 50000`
3. `SELECT * FROM employees WHERE salary > 50000;`
   `-- Employees with salary greater than 50000`
4. `SELECT * FROM employees WHERE salary < 50000;`
   `-- Employees with salary less than 50000`
5. `SELECT * FROM employees WHERE salary >= 50000;`
   `-- Employees with salary greater than or equal to 50000`
6. `SELECT * FROM employees WHERE salary <= 50000;`
   `-- Employees with salary less than or equal to 50000`

MORE USEFUL WHERE EXAMPLES

Query	What it Does
`SELECT * FROM employees WHERE department = 'IT';`	Employees from IT department
`SELECT * FROM employees WHERE position = 'Developer' AND salary > 50000;`	Developers with salary > 50000
`SELECT * FROM employees WHERE department = 'HR' OR salary >= 70000;`	HR department or salary >= 70000
`SELECT * FROM employees WHERE name LIKE 'A%';`	Names starting with A
`SELECT * FROM employees WHERE join_date >= '2024-01-15';`	Joined on or after 15 Jan 2024
`SELECT * FROM employees WHERE department IN ('IT','HR');`	Employees in IT or HR department
`SELECT * FROM employees WHERE salary BETWEEN 40000 AND 60000;`	Salary between 40000 and 60000
`SELECT * FROM employees WHERE position IS NULL;`	Position is NULL

KEY POINTS

- WHERE clause filters records based on a condition.
- Use comparison operators: =, !=, >, <, >=, <=
- Combine multiple conditions using AND, OR.
- Use LIKE for pattern matching and BETWEEN for ranges.
- Use IS NULL / IS NOT NULL to check NULL values.

VISUAL EXAMPLE

BEST PRACTICE

Always use WHERE clause to get only the data you need. It improves performance and makes queries faster.

QUICK REFERENCE

=	!=	>	<	>=	<=	WHERE
Equal to	Not equal to	Greater than	Less than	Greater than or equal to	Less than or equal to	Filter Data

TIP

Use proper indexes on columns used in WHERE clause for better performance.

ORDER BY

Sort and organize your results.

SELECT → ORDER BY → ASC / DESC → RESULTS

SELECT — Write Query
ORDER BY — Sort the Data
ASC / DESC — Choose Order
RESULTS — Sorted Output

WHAT IS ORDER BY?

ORDER BY clause is used to sort the result set in ascending (ASC) or descending (DESC) order by one or more columns.

1 ORDER BY CLAUSE

ORDER BY is used with **SELECT** statement to sort the result set by one or more columns.

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column_name;
```

SAMPLE TABLE: employees

id	name	department	salary	join_date
1	Alice	Developer	55000	2024-01-10
2	Bob	Designer	48000	2024-01-12
3	Charlie	Tester	42000	2024-01-15
4	Diana	HR	39000	2024-01-18
5	Eve	Developer	60000	2024-01-20
6	Frank	Manager	75000	2024-01-22
7	Grace	Intern	25000	2024-01-25

TYPES OF SORTING

Alphabetical (Text)	Numeric (Numbers)
A → Z	0 → 9
ASC : A to Z	ASC : Small to Large
DESC : Z to A	DESC : Large to Small

KEY TERMS

ORDER BY	Clause used to sort the result set.
Column	The column name used for sorting.
ASC	Ascending order (default).
DESC	Descending order.
Sorting	Arranging data in a specific order.
Results	The output after sorting.

2 ASC (Ascending Order)

ASC sorts the data in ascending order (default).

```
SELECT * FROM employees
ORDER BY salary ASC;
```

3 DESC (Descending Order)

DESC sorts the data in descending order.

```
SELECT * FROM employees
ORDER BY salary DESC;
```

QUICK TIP

If you don't specify ASC or DESC, ASC (Ascending) is used by default.

EXAMPLES WITH RESULTS

4 Sort by Column (ASC)

Sort by salary in ascending order.

```
SELECT * FROM employees
ORDER BY salary ASC;
```

Result

id	name	salary
7	Grace	25000
4	Diana	39000
3	Charlie	42000
2	Bob	48000
1	Alice	55000
5	Eve	60000
6	Frank	75000

5 Sort by Column (DESC)

Sort by salary in descending order.

```
SELECT * FROM employees
ORDER BY salary DESC;
```

Result

id	name	salary
6	Frank	75000
5	Eve	60000
1	Alice	55000
2	Bob	48000
3	Charlie	42000
4	Diana	39000
7	Grace	25000

6 Sort by Name (Alphabetical)

Sort by name in alphabetical order.

```
SELECT * FROM employees
ORDER BY name ASC;
```

Result

id	name
1	Alice
2	Bob
3	Charlie
4	Diana
5	Eve
6	Frank
7	Grace

7 Sort by Name (Z to A)

Sort by name in reverse alphabetical order.

```
SELECT * FROM employees
ORDER BY name DESC;
```

Result

id	name
7	Grace
6	Frank
5	Eve
4	Diana
3	Charlie
2	Bob
1	Alice

8 Multiple Columns

Sort by department ASC, then salary DESC.

```
SELECT * FROM employees
ORDER BY department ASC,
         salary DESC;
```

Result

department	name	salary
Developer	Eve	60000
Developer	Alice	55000
Designer	Bob	48000
HR	Diana	39000
Intern	Grace	25000
Manager	Frank	75000

ORDER BY SUMMARY

Use Case	Example
Sort A to Z	`ORDER BY name ASC`
Sort Z to A	`ORDER BY name DESC`
Small to Large	`ORDER BY salary ASC`
Large to Small	`ORDER BY salary DESC`
Sort by Date (Old to New)	`ORDER BY join_date ASC`
Sort by Date (New to Old)	`ORDER BY join_date DESC`
Sort by Multiple Columns	`ORDER BY dept ASC, salary DESC`

SYNTAX

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column_name1 ASC | DESC, column_name2 ASC | DESC, ...;
```

EXPLANATION

- You can sort by one column or multiple columns.
- Use **ASC** for ascending order (default).
- Use **DESC** for descending order.

BEST PRACTICE

Always use ORDER BY with LIMIT when working with large datasets to get meaningful results.

COMMON NOTES

- ASC is the default if not specified.
- ORDER BY comes after WHERE in a query.
- You can sort by column alias in some databases.

LIMIT

Control the number of rows returned.

WHAT IS LIMIT?

LIMIT clause is used to restrict the number of rows the query returns. Helps in pagination and better performance.

1 BASIC LIMIT

Return a specific number of rows.

```
SELECT * FROM employees
ORDER BY id ASC
LIMIT 5;
```

Result (First 5 Rows)

id	name	position	salary
1	Alice	Developer	55000
2	Bob	Designer	48000
3	Charlie	Tester	42000
4	Diana	HR	39000
5	Eve	Developer	60000

2 TOP (SQL Server)

Return top N rows. (Alternative to LIMIT)

```
SELECT TOP 5 * FROM employees
ORDER BY id ASC;
```

Result (Top 5 Rows)

id	name	position	salary
1	Alice	Developer	55000
2	Bob	Designer	48000
3	Charlie	Tester	42000
4	Diana	HR	39000
5	Eve	Developer	60000

3 LIMIT WITH OFFSET (Pagination)

Skip certain rows using OFFSET and then fetch limited rows.

```
SELECT * FROM employees
ORDER BY id ASC
LIMIT 5 OFFSET 5;
```

Explanation: Skip first 5 rows, then return next 5 rows.

Result (Rows 6 to 10)

id	name	position	salary
6	Frank	Manager	75000
7	Grace	Intern	25000
8	Henry	Analyst	67000
9	Ivy	Tester	41000
10	Jack	Developer	66000

4 PAGINATION FORMULA

General Pagination Query

```
SELECT * FROM employees
ORDER BY id ASC
LIMIT page_size OFFSET (page_number - 1) * page_size;
```

Example:

- **Page Size = 5**
- **Page Number = 3**

Query:

```
SELECT * FROM employees
ORDER BY id ASC
LIMIT 5 OFFSET (3 - 1) * 5;  -- OFFSET 10
```

Result: Rows 11 to 15

id	name	position	salary
11	Kate	HR	45000
12	Leo	Support	38000
13	Mia	Developer	62000
14	Noah	Designer	49000
15	Ollivia	Analyst	70000

KEY TERMS

LIMIT	Limits the number of rows returned.
TOP	Used in SQL Server to fetch top N rows.
OFFSET	Skips N rows before starting to return rows.
Pagination	Divides results into pages.
First Rows	The first N rows from the result set.
Records	Each row in the result set.
Fetch	Retrieve a limited number of rows.
Results	Output returned after applying LIMIT.

SUPPORT IN DATABASES

Database	LIMIT	OFFSET	TOP
MySQL	✅ Yes	✅ Yes	❌ No
PostgreSQL	✅ Yes	✅ Yes	❌ No
SQL Server	❌ No	❌ No	✅ Yes
Oracle 12c+	✅ Yes	✅ Yes	❌ No
SQLite	✅ Yes	✅ Yes	❌ No

5 USE CASES

Show Top Products	Recent Items	Pagination in Apps	Dashboard Widgets
Get top 10 best selling products.	Fetch last 5 latest records.	Display data page by page.	Show first N important data.

6 LIMIT – QUICK EXAMPLES (MySQL)

Query	Description
`SELECT * FROM employees LIMIT 3;`	First 3 rows
`SELECT * FROM employees LIMIT 10 OFFSET 5;`	Skip 5 rows, then get next 10
`SELECT * FROM employees LIMIT 1;`	Get only one row
`SELECT * FROM employees ORDER BY salary DESC LIMIT 3;`	Top 3 highest salaries
`SELECT * FROM employees LIMIT 5 OFFSET 20;`	Skip first 20, get next 5

7 SAMPLE DATA: employees

id	name	position	salary	join_date
1	Alice	Developer	55000	2024-01-10
2	Bob	Designer	48000	2024-01-12
3	Charlie	Tester	42000	2024-01-15
4	Diana	HR	39000	2024-01-18
...	...	...	...	...
23	Zack	Support	36000	2024-03-05
24	Yara	Analyst	68000	2024-03-07
25	Zoey	Developer	61000	2024-03-09

TIPS & BEST PRACTICES

- Always use ORDER BY with LIMIT for consistent results.
- Use LIMIT in dashboard queries to improve performance.
- Use OFFSET carefully in large tables; it can be slow for big offsets.
- Use proper indexing on columns used in ORDER BY for faster results.
- Use LIMIT for pagination, not for filtering data.

QUICK SYNTAX REFERENCE

```
-- Basic
SELECT * FROM table_name LIMIT N;
-- With Offset
SELECT * FROM table_name LIMIT N OFFSET M;
```

DISTINCT

Get Unique Results.

Return only unique (different) values from the result set.

SELECT Write Query → **DISTINCT** Remove Duplicates → **RESULTS** Unique Rows Returned

WHAT IS DISTINCT?

The DISTINCT keyword is used with SELECT statement to return only different (unique) values.

1 BASIC SYNTAX

Use DISTINCT to return unique values.

```
SELECT DISTINCT column_name
FROM table_name;
```

2 HOW IT WORKS

3 DISTINCT WITH MULTIPLE COLUMNS

DISTINCT works on the combination of columns.

```
SELECT DISTINCT column1, column2
FROM table_name;
```

Returns unique combinations of the selected columns.

SAMPLE TABLE: employees

id	name	department	city	salary
1	Alice	HR	Delhi	55000
2	Bob	IT	Mumbai	48000
3	Charlie	HR	Delhi	55000
4	Diana	IT	Bangalore	48000
5	Eve	HR	Pune	55000
6	Frank	IT	Mumbai	48000
7	Grace	HR	Delhi	60000
8	Henry	IT	Bangalore	48000

4 EXAMPLES WITH RESULTS

Query	All Rows (Original)	DISTINCT Results (Unique)
1 Unique values from one column `SELECT DISTINCT department FROM employees;`	HR, IT, HR, IT, HR, IT, HR, IT	HR IT
2 Unique cities `SELECT DISTINCT city FROM employees;`	Delhi, Mumbai, Delhi, Bangalore, Pune, Mumbai, Delhi, Bangalore	Delhi Mumbai Bangalore Pune
3 Unique salaries (Numeric) `SELECT DISTINCT salary FROM employees;`	55000, 48000, 55000, 48000, 55000, 48000, 60000, 48000	55000 48000 60000

KEY TERMS

DISTINCT	Returns only unique values
Unique	Different values
Duplicate	Repeated values
Rows	Each record in the table
Column	Field in a table
SELECT	SQL command to retrieve data
Filter	Used to remove duplicates
Results	Output after applying DISTINCT

DISTINCT vs ALL

Feature	DISTINCT	ALL (Default)
Returns	Unique values	All values (including duplicates)
Duplicates	Removed	Included
Default	No (must use keyword)	Yes
Performance	Slightly slower (duplicate check)	Faster

5 DISTINCT WITH MULTIPLE COLUMNS – EXAMPLE

```
SELECT DISTINCT department, city
FROM employees;
```

Returns unique combinations of department and city.

All Rows (Original)

department	city
HR	Delhi
IT	Mumbai
HR	Delhi
IT	Bangalore
HR	Pune
IT	Mumbai
HR	Delhi
IT	Bangalore

→ DISTINCT Results (Unique Combinations)

department	city
HR	Delhi
IT	Mumbai
IT	Bangalore
HR	Pune

6 USING WITH WHERE CLAUSE

```
SELECT DISTINCT city
FROM employees
WHERE salary > 50000;
```

Unique cities where salary is greater than $0000.

All Matching Rows

city	salary
Delhi	55000
Delhi	55000
Pune	55000
Delhi	60000

→ DISTINCT Results

city
Delhi
Pune

7 USING WITH ORDER BY

```
SELECT DISTINCT city
FROM employees
ORDER BY city ASC;
```

Unique cities sorted alphabetically.

DISTINCT + ORDER BY Result

city
Bangalore
Delhi
Mumbai
Pune

8 COUNT DISTINCT

Count unique values.

```
SELECT COUNT(DISTINCT city)
FROM employees;
```

Counts the number of unique cities.

Result
4

QUICK TIPS

- Use DISTINCT to get clean, unique data.
- Works with text, numbers, dates, and multiple columns.
- DISTINCT is often used with WHERE, ORDER BY, COUNT().

BEST PRACTICES

- Select only the columns you need with DISTINCT.
- Use DISTINCT with WHERE to filter before removing duplicates.
- Use DISTINCT to improve reports and summaries.

COMMON MISTAKES

- Forgetting DISTINCT when duplicates appear.
- Using DISTINCT on unnecessary columns.
- DISTINCT may slow down on very large datasets.

SUMMARY

DISTINCT returns only unique values.
It removes duplicate rows from the result set, helping you get clean and meaningful data.

UPDATE DATA

Modify existing data in a table.

UPDATE is used to change one or more columns of existing records.

HOW UPDATE WORKS

UPDATE → SET → WHERE → SAVE

- **UPDATE** — Specify table
- **SET** — Set new value
- **WHERE** — Filter the record
- **SAVE** — Changes are saved

WHAT IS UPDATE?

UPDATE statement is used to modify existing data in a table.

Always use WHERE clause to update specific records safely.

KEY TERMS

UPDATE	SQL command used to modify data.
SET	Specifies the column(s) and the new value.
WHERE	Filters the record(s) to be updated.
Modify	Make changes to existing data.
Record	A single row in a table.
Column	A field/attribute in a table.
Value	The data stored in a column.
Save	Changes are committed and stored.

BASIC SYNTAX

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

⚠ Without **WHERE** clause, all records will be updated!

SAMPLE TABLE: employees

id	name	department	salary	city
1	Alice	HR	55000	Delhi
2	Bob	IT	48000	Mumbai
3	Charlie	IT	52000	Pune
4	Diana	HR	45000	Delhi
5	Eve	Finance	60000	Bangalore

BEFORE vs AFTER UPDATE

BEFORE UPDATE

id	name	department	salary	city
2	Bob	IT	48000	Mumbai

AFTER UPDATE

id	name	department	salary	city
2	Bob	IT	55000	Mumbai

EXAMPLES

1 Update a single column

Increase salary of employee with id = 2.

```
UPDATE employees
SET salary = 55000
WHERE id = 2;
```

Result:

id	name	department	salary	city
2	Bob	IT	55000	Mumbai

2 Update multiple columns

Change department and city of employee with id = 3.

```
UPDATE employees
SET department = 'Data Science', city = 'Hyderabad'
WHERE id = 3;
```

Result:

id	name	department	salary	city
3	Charlie	Data Science	52000	Hyderabad

3 Update with condition

Increase salary by 5000 for all employees in IT department.

```
UPDATE employees
SET salary = salary + 5000
WHERE department = 'IT';
```

Result: (Only IT department updated)

id	name	department	salary	city
2	Bob	IT	53000	Mumbai
3	Charlie	IT	57000	Pune

ANATOMY OF UPDATE

```
UPDATE employees SET salary = 55000 WHERE id = 2;
```

- `UPDATE` — Command to update data
- `employees` — Table name
- `SET salary = 55000` — Column to be set with new value
- `WHERE id = 2` — Condition to find the record

BEST PRACTICES

- Always use WHERE clause to avoid updating all records.
- Test your UPDATE query with SELECT first.
- Use meaningful conditions for accurate results.
- Backup your data before running bulk updates.

COMMON MISTAKES

- Forgetting WHERE clause (updates all rows).
- Wrong condition (updates wrong records).
- Setting wrong value or column name.
- Not checking the result after update.

QUICK EXAMPLE – SELECT BEFORE UPDATE

Check Current Data

```
SELECT * FROM employees
WHERE id = 4;
```

id	name	department	salary	city
4	Diana	HR	45000	Delhi

After Update

```
UPDATE employees
SET salary = 47000
WHERE id = 4;
```

id	name	department	salary	city
4	Diana	HR	47000	Delhi

QUICK SUMMARY

- UPDATE – Command to modify data.
- SET – Column and new value.
- WHERE – Filter the record(s).
- Modify – Change the existing data.
- Save – Changes are stored permanently.

PRO TIP: Always run SELECT query first to confirm the data, then use UPDATE to modify, and verify again.

SELECT → UPDATE → SELECT

Safe Update = Smart Update!

DELETE DATA

Remove unwanted data from a table.

DELETE statement is used to remove existing rows (records) from a table based on a condition.

HOW DELETE WORKS

DELETE — Write Query → **WHERE** — Set condition → **REMOVE** — Matching rows are removed → **RESULT** — Remaining rows stay safe

WHAT IS DELETE?

DELETE statement removes one or more rows from a table permanently.

Always use WHERE clause to delete specific records safely.

KEY TERMS

DELETE	SQL command used to remove data.
WHERE	Filters the rows to be deleted.
Remove	Deletes matching rows from the table.
Rows	Horizontal data entries in a table.
Records	A complete row of related data.
Table	A collection of related data in row-column format.
Safe Delete	Using WHERE clause to avoid accidental deletion.
Query	SQL statement written to perform delete operation.

BASIC SYNTAX

```
DELETE FROM table_name
WHERE condition;
```

⚠ Without WHERE clause, all rows in the table will be deleted!

SAMPLE TABLE: employees

id	name	department	salary	city
1	Alice	HR	55000	Delhi
2	Bob	IT	48000	Mumbai
3	Charlie	IT	52000	Pune
4	Diana	Finance	60000	Bangalore
5	Eve	HR	45000	Delhi
6	Frank	IT	47000	Hyderabad

BEFORE vs AFTER DELETE

BEFORE DELETE

id	name	department	salary	city
1	Alice	HR	55000	Delhi
2	Bob	IT	48000	Mumbai
3	Charlie	IT	52000	Pune
4	Diana	Finance	60000	Bangalore
5	Eve	HR	45000	Delhi
6	Frank	IT	47000	Hyderabad

AFTER DELETE `(department = 'IT')`

id	name	department	salary	city
1	Alice	HR	55000	Delhi
4	Diana	Finance	60000	Bangalore
5	Eve	HR	45000	Delhi

EXAMPLES

1 Delete a specific record

Delete employee with id = 3

```
DELETE FROM employees
WHERE id = 3;
```

Result:

id	name	department	salary	city
1	Alice	HR	55000	Delhi
2	Bob	IT	48000	Mumbai
4	Diana	Finance	60000	Bangalore
5	Eve	HR	45000	Delhi
6	Frank	IT	47000	Hyderabad

2 Delete multiple records

Delete all employees from HR department

```
DELETE FROM employees
WHERE department = 'HR';
```

Result:

id	name	department	salary	city
2	Bob	IT	48000	Mumbai
3	Charlie	IT	52000	Pune
4	Diana	Finance	60000	Bangalore
6	Frank	IT	47000	Hyderabad

3 Delete with condition

Delete employees with salary less than 50000

```
DELETE FROM employees
WHERE salary < 50000;
```

Result:

id	name	department	salary	city
1	Alice	HR	55000	Delhi
3	Charlie	IT	52000	Pune
4	Diana	Finance	60000	Bangalore

4 Delete all records (Use Carefully!)

Delete all rows from the table

```
DELETE FROM employees;
```

⚠ All rows will be deleted. Table structure will remain. (Very dangerous in real apps)

SAFE DELETE CHECKLIST

- ✓ Always use WHERE clause.
- ✓ Double-check your condition.
- ✓ Run SELECT query first to preview rows.
- ✓ Use transactions for important deletions.
- ✓ Backup data before performing bulk delete.

PRACTICE: SELECT BEFORE DELETE

Always run SELECT first to see which rows will be deleted.

```
SELECT * FROM employees
WHERE department = 'IT';
```

→ Check the result. If correct, then run DELETE query.

COMMON MISTAKES

- ✗ Using DELETE without WHERE clause.
- ✗ Wrong condition (may delete wrong data).
- ✗ Not checking the data before delete.
- ✗ Not using transactions for bulk delete.

QUICK SUMMARY

Action	SQL Command
Delete specific row	`DELETE FROM table WHERE id = 1;`
Delete with condition	`DELETE FROM table WHERE condition;`
Delete all rows	`DELETE FROM table;`

PRO TIP

Combine DELETE with WHERE, AND, OR, IN, BETWEEN for powerful and precise data removal.

Example: `DELETE FROM employees WHERE city = 'Delhi' AND salary < 50000;`

IMPORTANT NOTE

DELETE removes data **PERMANENTLY.**
There is no UNDO. Be 100% sure before running.

DELETE WORKFLOW

Write DELETE Query → Test with SELECT → Confirm Condition → Run DELETE → Verify Results

SAMPLE QUERY

```
DELETE FROM employees
WHERE city = 'Mumbai' AND salary < 50000;
```

TRUNCATE & DROP

Powerful SQL Commands to Remove Data or Structure

DELETE ALL — Remove all rows at once

RESET DATA — Quick way to reset table

REMOVE TABLE — Delete a table completely

DROP DATABASE — Remove entire database

PERMANENT — Changes are final and cannot be undone

KEY DIFFERENCE AT A GLANCE

Feature	TRUNCATE	DROP TABLE	DROP DATABASE
Deletes Data	Yes (All Rows)	Yes (Table + Data)	Yes (All Tables + Data)
Structure Removed	No	Yes	Yes
Reset / Free Space	Yes	Yes	Yes
Rollback Possible	No	No	No
Permanent	Yes	Yes	Yes

1) TRUNCATE TABLE

Removes ALL rows from a table, but keeps the table structure.
It is faster than DELETE and resets identity/auto-increment values.

SYNTAX

```
TRUNCATE TABLE table_name;
```

EXAMPLE

```
TRUNCATE TABLE employees;
```

EFFECT

- Deletes all rows
- Keeps table structure
- Resets identity values
- Faster performance
- Cannot be rolled back

BEFORE vs AFTER – TRUNCATE TABLE

BEFORE (employees)

id	name	department	salary
1	Alice	HR	55000
2	Bob	IT	48000
3	Charlie	Finance	52000

→ **AFTER TRUNCATE (employees)**

0 ROWS
(Table structure remains)

USE WHEN:
You want to delete all data from a table but keep the table for future use.

NOTES:
- Cannot use WHERE clause.
- Implicit commit in most DBMS.

2) DROP TABLE

Removes the entire table including its structure, indexes, constraints and all data permanently.

SYNTAX

```
DROP TABLE table_name;
```

EXAMPLE

```
DROP TABLE employees;
```

EFFECT

- Deletes all rows
- Removes table structure
- Removes indexes, constraints, triggers
- Space is freed
- Cannot be rolled back

BEFORE vs AFTER – DROP TABLE

BEFORE (employees)

id	name	department	salary
1	Alice	HR	55000
2	Bob	IT	48000
3	Charlie	Finance	52000

→ **AFTER DROP TABLE**

TABLE DOES NOT EXIST
(Structure and data removed)

USE WHEN:
You no longer need the table and want to remove it completely.

WARNING:
This action is permanent and cannot be undone.

3) DROP DATABASE

Removes the entire database including all tables, data, indexes, and objects permanently.

SYNTAX

```
DROP DATABASE database_name;
```

EXAMPLE

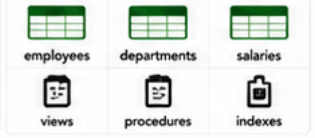
```
DROP DATABASE company_db;
```

EFFECT

- Deletes all tables
- Deletes all data
- Deletes all objects (views, procedures, functions, etc.)
- Frees all space
- Cannot be rolled back

BEFORE vs AFTER – DROP DATABASE

BEFORE (company_db)



→ **AFTER DROP DATABASE**

DATABASE DOES NOT EXIST
(Everything removed)

USE WHEN:
You no longer need the entire database and want to remove it completely.

EXTREME WARNING:
This is a destructive command. Double-check before running!

QUICK COMPARISON

Command	Deletes Data	Structure	Resets Identity	WHERE Clause	Rollback	Use Case
TRUNCATE TABLE	All Rows	Kept	Yes	Not Allowed	No	Reset/clear data, keep table
DROP TABLE	All Rows	Removed	N/A	N/A	No	Remove table completely
DROP DATABASE	All Data	Removed	N/A	N/A	No	Remove entire database

IMPORTANT POINTS

- All three commands are DDL (Data Definition Language).
- They cause an implicit commit in most databases.
- These operations are PERMANENT – use with extreme care!
- Always backup your data before performing these commands.

SAMPLE QUERIES

```
-- Truncate table
TRUNCATE TABLE employees;

-- Drop table
DROP TABLE employees;

-- Drop database
DROP DATABASE company_db;
```

PRO TIP

When in doubt,
don't drop!

There is no UNDO
for these commands.

SQL CONSTRAINTS

Rules for Accurate, Reliable and Consistent Data

Constraints are used to enforce rules on data in a table.
They help maintain data integrity and consistency.

WHY USE CONSTRAINTS?

- **Data Integrity** — Prevent invalid or duplicate data
- **Accuracy** — Ensure correct data entry
- **Consistency** — Maintain reliable relationships
- **Security** — Protect important data
- **Performance** — Faster queries with indexes

1 PRIMARY KEY

Uniquely identifies each row in a table.
No duplicate values.
Cannot be NULL.

SYNTAX

```
column_name DATA_TYPE
PRIMARY KEY
```

EXAMPLE (Employees table)

EmpID	Name	Email
101	Alice	alice@mail.com
102	Bob	bob@mail.com
103	Charlie	charlie@mail.com

EXAMPLE SQL

```
CREATE TABLE Employees (
  EmpID INT PRIMARY KEY,
  Name VARCHAR(50),
  Email VARCHAR(100)
);
```

2 FOREIGN KEY

Creates a relationship between two tables.
Ensures referential integrity.

SYNTAX

```
column_name DATA_TYPE
FOREIGN KEY (column_name)
REFERENCES table_name(column_name)
```

EXAMPLE

Orders table referencing Employees table

OrderID	EmpID	OrderDate
1	101	2024-05-01
2	102	2024-05-02
3	101	2024-05-03

EXAMPLE SQL

```
CREATE TABLE Orders (
  OrderID INT PRIMARY KEY,
  EmpID INT,
  OrderDate DATE,
  FOREIGN KEY (EmpID)
    REFERENCES Employees(EmpID)
);
```

3 UNIQUE

Ensures all values in a column (or set of columns) are unique.
Allows NULL values.

SYNTAX

```
column_name DATA_TYPE
UNIQUE
```

EXAMPLE (Users table)

UserID	Username	Email
1	jsmith	jsmith@mail.com
2	pmiller	pmiller@mail.com
3	adavis	adavis@mail.com

EXAMPLE SQL

```
CREATE TABLE Users (
  UserID INT PRIMARY KEY,
  Username VARCHAR(50) UNIQUE,
  Email VARCHAR(100) UNIQUE
);
```

4 NOT NULL

Ensures a column cannot have NULL value.
A value is required.

SYNTAX

```
column_name DATA_TYPE
NOT NULL
```

EXAMPLE (Employees table)

EmpID	Name	Salary
101	Alice	50000
102	Bob	45000
103	Charlie	55000

EXAMPLE SQL

```
CREATE TABLE Employees (
  EmpID INT PRIMARY KEY,
  Name VARCHAR(50) NOT NULL,
  Salary DARCMAL(10,2) NOT NULL
);
```

5 DEFAULT

Sets a default value for a column when no value is specified.

SYNTAX

```
column_name DATA_TYPE
DEFAULT default_value
```

EXAMPLE (Products table)

ProdID	ProductName	Status
1	Laptop	Active
2	Mobile	Active
3	Tablet	Active

EXAMPLE SQL

```
CREATE TABLE Empćuets (
  ProdID INT PRIMARY KEY,
  ProductName VARCHAR(100),
  Status VARCHAR(20)
  DEFAULT 'Active'
);
```

6 CHECK

Ensures all values in a column satisfy a specific condition.

SYNTAX

```
column_name DATA_TYPE
CHECK (condition)
```

EXAMPLE (Employees table)

EmpID	Age	Salary
101	25	50000
102	30	45000
103	28	55000

EXAMPLE SQL

```
CREATE TABLE Employees (
  EmpID INT PRIMARY KEY,
  Age INT CHECK (Age >= 18)
      AND Age <= 60),
  Salary DECIMAL(10,2)
  CHECK (Salary > 0)
);
```

7 INDEX

Improves the speed of data retrieval operations.
(Not a constraint, but often used with constraints.)

SYNTAX

```
CREATE INDEX index_name
ON table_name(column_name);
```

EXAMPLE (Employees table)

EmpID	Name	Email
101	Alice	alice@mail.com
102	Bob	bob@mail.com
103	Charlie	charlie@mail.com

EXAMPLE SQL

```
CREATE INDEX idx_emp_name
ON Employees(Name);

CREATE UNIQUE INDEX idx_emp_email
ON Employees(Email);
```

CONSTRAINTS IN ACTION – RELATIONSHIP EXAMPLE

Employees Table (Parent)

EmpID	Name	Department
101	Alice	HR
102	Bob	IT
103	Charlie	Finance

EmpID is **PRIMARY KEY** in Employees →

Orders Table (Child)

OrderID	EmpID	OrderDate	Amount
1	101	2024-05-01	1200
2	102	2024-05-02	1500
3	101	2024-05-03	1000

FOREIGN KEY
Orders.EmpID is FOREIGN KEY referencing Employees.EmpID

QUICK SUMMARY TABLE

Constraint	Ensures	Allows NULL	Allows Duplicates	Purpose
PRIMARY KEY	Unique + Not NULL	No	No	Unique row identification
FOREIGN KEY	Valid reference	Yes*	Yes	Maintain relationship
UNIQUE	Unique values	Yes	No	Prevent duplicates
NOT NULL	Value must exist	No	Yes	Mandatory data
DEFAULT	Default value if none	Yes	Yes	Set default data
CHECK	Valid per condition	Yes	Yes	Validate data
INDEX	Fast data access	Yes	Yes	Improve performance

* In Some DBMS, FOREIGN KEY column can be NULL.

BEST PRACTICES

- Always define **PRIMARY KEY** for each table.
- Use **FOREIGN KEY** to maintain relationships.
- Use **NOT NULL** for important columns.
- Use **CHECK** to validate data at the database level.
- Use **INDEX** on columns used frequently in search/filter.

COMMON ERRORS

- Forgetting **PRIMARY KEY**.
- Inserting **NULL** in **NOT NULL** column.
- Duplicate values in **UNIQUE** column.
- Violating **CHECK** condition.
- Deleting parent record with existing child (FK).

TIPS

- Constraints protect your data.
- They reduce errors from application side.
- Good constraints = Good Database Design.
- Always test constraints with real data.

NOTE

Constraints make your database strong, reliable and consistent.
Use them wisely for better data quality and performance!

PRIMARY KEY

Unique Identifier for Every Row

A PRIMARY KEY uniquely identifies each row in a table.
It must contain **unique values** and cannot be **NULL**.

IN SHORT

Unique (No duplicates) + **Auto Increment** (Automatic value) + **Identity** (Built-in identity) + **Row ID** (Unique Row ID) = **Primary Key** (Perfect Identifier)

KEY CONCEPTS

- **UNIQUE** — Each value in the primary key must be unique.
- **AUTO INCREMENT** — Automatically increases the value for each new row.
- **IDENTITY** — System generated value (Identity Column).
- **ROW ID** — Acts as a unique row identifier.
- **TABLE** — Primary key belongs to a table and identifies rows.
- **INDEX** — Primary key creates a unique clustered index automatically.
- **CONSTRAINT** — Primary key is a constraint that enforces uniqueness and NOT NULL.

1) WHAT IS PRIMARY KEY?

A **PRIMARY KEY** is a column (or combination of columns) that uniquely identifies each row in a table.
It enforces **UNIQUE** and **NOT NULL** values.

Example Table: Employees

EmpID (PK)	Name	Department	Salary
1	Alice	HR	55000
2	Bob	IT	48000
3	Charlie	Finance	52000
4	Diana	IT	60000

EmpID is the **PRIMARY KEY**. It uniquely identifies each employee.

2) SYNTAX

Method 1: While Creating Table

```
CREATE TABLE Employees (
  EmpID  INT PRIMARY KEY,
  Name   VARCHAR(50),
  Department   VARCHAR(50),
  Salary DECIMAL(10,2)
);
```

Method 2: With Auto Increment / Identity

```
CREATE TABLE Employees (
  EmpID  INT  IDENTITY(1,1) PRIMARY KEY,
  Name   VARCHAR(50),
  Department  VARCHAR(50),
  Salary DECIMAL(10,2)
);
```

Auto Increment (Starts from 1, increases by 1)

3) PROPERTIES OF PRIMARY KEY

- Must contain UNIQUE values.
- Cannot contain NULL values.
- Only one primary key per table.
- Can be single column or composite (multi-column).
- Automatically creates a UNIQUE CLUSTERED INDEX.
- Helps in fast data retrieval.

4) PRIMARY KEY CREATES INDEX

Table (Employees) → Primary Key creates → Unique Clustered Index (on EmpID)

This index ensures fast lookups and uniqueness.

5) EXAMPLE: HOW PRIMARY KEY WORKS

Valid Insert

EmpID	Name	Department
1	Alice	HR
2	Bob	IT
3	Charlie	Finance

Invalid Insert (Will Fail)

EmpID	Name	Department	
2	David	HR	→ Duplicate EmpID
NULL	Eve	IT	→ NULL Not Allowed

Duplicate or NULL values are not allowed in PRIMARY KEY.

6) COMPOSITE PRIMARY KEY (MULTI-COLUMN)

When a combination of columns uniquely identifies a row.

```
CREATE TABLE OrderItems (
  OrderID  INT,
  ProductID  INT,
  Quantity INT,
  PRIMARY KEY (OrderID, ProductID)
);
```

OrderItems Table

OrderID	ProductID	Quantity
101	1	2
101	2	1
102	1	3

Here, (OrderID, ProductID) together form the PRIMARY KEY.

7) PRIMARY KEY vs UNIQUE KEY

Feature	PRIMARY KEY	UNIQUE KEY
Allows NULL	No	Yes (One NULL allowed)
Allows Duplicates	No	No
Only One per Table	Yes	No (Multiple allowed)
Creates Clustered Index	Yes	No (Creates Non-clustered Index)
Used For	Row Identification	Ensure Uniqueness

BEST PRACTICES

- Always use meaningful primary keys.
- Use **Auto Increment/Identity** for better performance.
- Avoid using large text or random columns as PK.
- Use composite key only when necessary.

TIPS

Primary key is best for relationships between tables.
It improves data accuracy and performance.
Always index your primary key.

COMMON MISTAKES

- Using NULL in primary key.
- Using duplicate values.
- Changing primary key values frequently.
- Not choosing the right column as primary key.

QUICK SUMMARY

- PRIMARY KEY = Unique + Not NULL
- Auto Increment makes it easy.
- Creates unique clustered index.
- Identifies each row in a table.

FOREIGN KEY

Link Tables. Maintain Relationships. Ensure Integrity.

A FOREIGN KEY is a column (or set of columns) in a child table that refers to the **PRIMARY KEY** of a parent table.

Relationship — Creates bond between tables → **Parent Table** — Table being referenced → **Child Table** — Table that holds the foreign key → **Reference** — References the primary key → **Integrity** — Preserves referential integrity → **Join / Link** — Used to fetch related data

KEY CONCEPTS

- **RELATIONSHIP** — Foreign key creates relationship between two tables.
- **PARENT TABLE** — The table that has the PRIMARY KEY (referenced table).
- **CHILD TABLE** — The table that has the FOREIGN KEY (referencing table).
- **REFERENCE** — Foreign key refers to the primary key of parent table.
- **INTEGRITY** — Ensures valid data. No orphan records in child table.
- **JOIN** — Used to combine data from related tables.
- **LINK** — Helps to link data and build normalized databases.

1) EXAMPLE RELATIONSHIP

One Department can have many Employees.
Department is Parent, Employee is Child.

Parent Table: Departments

DeptID (PK)	DeptName
1	HR
2	IT
3	Finance

One to Many (1 → ∞)

Child Table: Employees

EmpID (PK)	EmpName	DeptID (FK)
101	Alice	1
102	Bob	2
103	Charlie	1
104	Diana	3

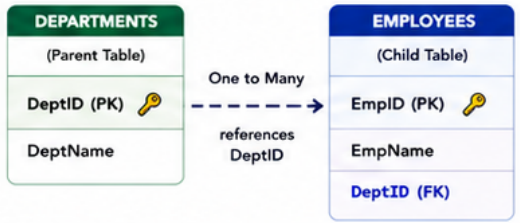
Employees.DeptID is a **FOREIGN KEY** that references **Departments.DeptID**

2) CREATE FOREIGN KEY – SYNTAX

```
CREATE TABLE ChildTable (
  Column1 datatype,
  Column2 datatype,
  ...,
  ForeignKeyColumn datatype,
  CONSTRAINT fk_name FOREIGN KEY (ForeignKeyColumn)
     REFERENCES ParentTable(PrimaryKeyColumn)
     ON UPDATE CASCADE
     ON DELETE RESTRICT
);
```

- `CONSTRAINT fk_name` → Name of the foreign key
- `REFERENCES` → Parent table and its primary key
- `ON UPDATE` → Action when parent key is updated
- `ON DELETE` → Action when parent row is deleted
- (CASCADE | SET NULL | RESTRICT | NO ACTION)

3) RELATIONSHIP DIAGRAM



PK = Primary Key
FK = Foreign Key
----- = Relationship Link

4) EXAMPLE – CREATE TABLES WITH FOREIGN KEY

Parent Table Departments

```
CREATE TABLE Departments (
  DeptID INT PRIMARY KEY,
  DeptName VARCHAR(50) NOT NULL
);

INSERT INTO Departments VALUES
(1,'HR'), (2,'IT'), (3,'Finance');
```

Child Table Employees

```
CREATE TABLE Employees (
  EmpID INT PRIMARY KEY,
  EmpName VARCHAR(50) NOT NULL,
  DeptID INT,
  CONSTRAINT fk_emp_dept FOREIGN KEY (DeptID)
     REFERENCES Departments(DeptID)
     ON UPDATE CASCADE
     ON DELETE RESTRICT
);

INSERT INTO Employees VALUES
(101,'Alice',1), (102,'Bob',2), (103,'Charlie',1);
```

5) REFERENTIAL INTEGRITY (WHAT IT DOES)

ALLOWED (Valid)

- Inserting a child row with existing DeptID.
- Updating DeptID if parent key is updated (CASCADE).
- Deleting parent row if no child rows exist.

NOT ALLOWED (Invalid)

- Inserting a child row with non-existing DeptID.
- Deleting parent row if child rows exist (RESTRICT).
- Leaves orphan records (protected).

Referential integrity keeps your data accurate and consistent.

6) JOIN EXAMPLE – GET RELATED DATA

```
SELECT e.EmpID, e.EmpName, d.DeptName
FROM Employees e
INNER JOIN Departments d
   ON e.DeptID = d.DeptID;
```

Result

EmpID	EmpName	DeptName
101	Alice	HR
102	Bob	IT
103	Charlie	HR
104	Diana	Finance

JOIN uses the **FOREIGN KEY** to link data between tables.

BEST PRACTICES

- Always define FOREIGN KEY to enforce relationships.
- Index foreign key columns for better performance.
- Use meaningful constraint names.
- Choose proper ON DELETE / ON UPDATE actions.
- Test relationships before inserting or deleting data.

ON DELETE OPTIONS

- **CASCADE** → Delete child rows automatically.
- **SET NULL** → Set child foreign key to NULL.
- **RESTRICT** → Prevent delete if child rows exist.
- **NO ACTION** → Same as RESTRICT (standard).

ON UPDATE OPTIONS

- **CASCADE** → Update child keys automatically.
- **SET NULL** → Set child foreign key to NULL.
- **RESTRICT** → Prevent update if child rows exist.
- **NO ACTION** → Same as RESTRICT (standard).

COMMON MISTAKES

- Referencing non-unique column.
- Data type mismatch between keys.
- Forgetting to index foreign keys.
- Deleting parent rows without rule.

QUICK SUMMARY

- FOREIGN KEY creates relationship.
- References PRIMARY KEY of parent.
- Ensures data integrity.
- Used for JOIN and linking data.

ALTER TABLE

Modify the Structure (Schema) of an Existing Table

ALTER TABLE is used to add, delete, or modify columns, rename tables/columns, or manage constraints.

WHAT ALTER TABLE CAN DO?

- **ADD COLUMN** — Add new column(s)
- **DROP COLUMN** — Remove column(s)
- **MODIFY** — Change data type, size or definition
- **RENAME** — Rename table or column
- **CONSTRAINT** — Add/Drop constraints
- **SCHEMA** — Update table structure
- **UPDATE** — Apply changes safely

KEY CONCEPTS

- **ADD COLUMN** — Add a new column to the table.
- **DROP COLUMN** — Remove an existing column from the table.
- **MODIFY** — Change the data type, size, or default value of a column.
- **RENAME** — Rename a table or a column.
- **CONSTRAINT** — Add, modify or drop constraints (PK, FK, UNIQUE, CHECK, etc.)
- **SCHEMA** — Alter the structure of the table.
- **UPDATE** — Update the structure without losing data.

NOTE
ALTER TABLE changes the structure, not the data. Data remains safe.

EXAMPLE TABLE (BEFORE ALTER)

Table Name: Employees

EmpID (PK)	Name	Department	Salary
101	Alice	HR	50000
102	Bob	IT	45000
103	Charlie	Finance	55000

Current Columns:
- EmpID
- Name
- Department
- Salary

1) ADD COLUMN

Add a new column to the existing table.

SYNTAX

```
ALTER TABLE table_name
ADD COLUMN column_name datatype
[DEFAULT value];
```

EXAMPLE

```
ALTER TABLE Employees
ADD COLUMN Email VARCHAR(100);
```

RESULT (AFTER ADD COLUMN)

EmpID	Name	Department	Salary	Email
101	Alice	HR	50000	NULL
102	Bob	IT	45000	NULL
103	Charlie	Finance	55000	NULL

2) DROP COLUMN

Remove an existing column from the table.

SYNTAX

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

EXAMPLE

```
ALTER TABLE Employees
DROP COLUMN Email;
```

RESULT (AFTER DROP COLUMN)

EmpID	Name	Department	Salary
101	Alice	HR	50000
102	Bob	IT	45000
103	Charlie	Finance	55000

3) MODIFY COLUMN

Change data type, size, or default value of a column.

SYNTAX

```
ALTER TABLE table_name
MODIFY COLUMN column_name
new_datatype [DEFAULT value];
```

EXAMPLE

```
ALTER TABLE Employees
MODIFY COLUMN Salary DECIMAL(10,2);
```

RESULT (AFTER MODIFY COLUMN)

EmpID	Name	Department	Salary (DECIMAL)
101	Alice	HR	50000.00
102	Bob	IT	45000.00
103	Charlie	Finance	55000.00

4) RENAME

Rename a table or a column.

Rename Column

SYNTAX

```
ALTER TABLE table_name
RENAME COLUMN old_name TO new_name;
```

EXAMPLE

```
ALTER TABLE Employees
RENAME COLUMN Name TO FullName;
```

Rename Table

SYNTAX

```
ALTER TABLE old_table_name
RENAME TO new_table_name;
```

EXAMPLE

```
ALTER TABLE Employees
RENAME TO Staff;
```

5) CONSTRAINT

Add or drop constraints to maintain rules.

Add Constraint (Example)

```
ALTER TABLE Employees
ADD CONSTRAINT chk_salary
CHECK (Salary > 0);
```

Drop Constraint (Example)

```
ALTER TABLE Employees
DROP CONSTRAINT chk_salary;
```

Add Foreign Key (Example)

```
ALTER TABLE Employees
ADD CONSTRAINT fk_dept
FOREIGN KEY (DeptID)
REFERENCES Departments(DeptID);
```

6) SCHEMA UPDATE EXAMPLE

You can update multiple elements of table structure.

EXAMPLE

```
ALTER TABLE Employees
 ADD COLUMN JoinDate DATE,
 MODIFY COLUMN Salary DECIMAL(10,2),
 DROP COLUMN OldField;
```

RESULT (SCHEMA AFTER ALTERATIONS)

EmpID	Name	Department	Salary	JoinDate
101	Alice	HR	50000.00	2024-01-15
102	Bob	IT	45000.00	2024-02-20
103	Charlie	Finance	55000.00	2024-03-10

QUICK SUMMARY

Action	Description	Syntax (Example)	Affects Data?	Common Use
ADD COLUMN	Add a new column	`ALTER TABLE t ADD COLUMN c type;`	No	Add new info
DROP COLUMN	Remove a column	`ALTER TABLE t DROP COLUMN c;`	No (data in column lost)	Remove unused info
MODIFY	Change column type/size/default	`ALTER TABLE t MODIFY COLUMN c type;`	No (structure only)	Change definition
RENAME COLUMN	Rename a column	`ALTER TABLE t RENAME COLUMN old TO new;`	No	Better naming
RENAME TABLE	Rename a table	`ALTER TABLE old_name RENAME TO new_name;`	No	Table rename
CONSTRAINT	Add/Drop constraints	`ALTER TABLE t ADD/DROP CONSTRAINT ...;`	No (structure only)	Maintain rules

BEST PRACTICES

- Always take a backup before altering tables.
- Test changes in a development environment.
- Use meaningful names for columns and constraints.
- Check dependencies before dropping columns.
- Use transactions for critical schema changes.

COMMON ERRORS

- Dropping a column used in views or queries.
- Modifying column to incompatible data type.
- Forgetting to update dependent objects.
- Adding NOT NULL column without DEFAULT.
- Violating constraints while altering structure.

IMPORTANT NOTES

- ALTER TABLE only changes the structure (schema).
- Existing data remains safe.
- Use constraints to maintain data integrity.
- Some changes may lock the table temporarily.

JOIN EXAMPLE (USING FOREIGN KEY)

Employees.DeptID references Departments.DeptID

```
SELECT e.EmpID, e.Name, d.DeptName
FROM Employees e
INNER JOIN Departments d
ON e.DeptID = d.DeptID;
```

SQL COMMENTS

Write Notes in Your Code, Make It Easy to Understand!

SQL Comments are used to add explanations or notes in SQL code. They are ignored by the database engine and help in **documentation, readability** and **teamwork.**

WHY USE SQL COMMENTS?

- **DOCUMENTATION** — Explain what the code does
- **READABLE** — Make code easy to read
- **NOTES** — Add important notes or ideas
- **MAINTAIN CODE** — Easier to maintain and debug
- **TEAMWORK** — Help others understand your code

1) SINGLE LINE COMMENTS

`--` Starts with double hyphen (--).
It comments out the rest of the line.

SYNTAX

```
-- This is a single line comment
```

EXAMPLE

```
SELECT EmpID, Name, Salary  -- Get employee details
FROM Employees;

SELECT * FROM Employees;  -- Select all columns
-- This query is used to check employee data
WHERE Salary > 50000;    -- High salary employees
```

OUTPUT BEHAVIOR

The database ignores everything after -- on that line.

2) MULTI LINE COMMENTS

`/*` `*/`

Starts with /* and ends with */.
It can span multiple lines.

SYNTAX

```
/*
  This is a
  multi line comment
*/
```

EXAMPLE

```
/*
This query will fetch
employee ID, Name and Department
from the Employees table
where salary is greater than 50000.
*/
SELECT EmpID, Name, Department
FROM Employees
WHERE Salary > 50000;
```

OUTPUT BEHAVIOR

The database ignores everything between /* and */.

QUICK REMINDER

--	Single Line Comment	Comments till end of the line
/* */	Multi Line Comment	Comments across multiple lines
</>	Ignored by SQL Engine	Not executed, only for humans
✓	Improves Readability	Makes code clean and easy to understand
	Use Meaningful Notes	Write clear and helpful comments

3) REAL WORLD EXAMPLE

```
-- ================================
-- Employees Report Query
-- Author: Lovejeet Singh
-- Purpose: Get active employees with high salary
-- Date: 25 May 2025
-- ================================

SELECT EmpID, Name, Department, Salary  -- Select required columns
FROM Employees
WHERE IsActive = 1                -- Only active employees
AND  Salary > 50000;              -- High salary filter
```

4) WITH COMMENTS vs WITHOUT COMMENTS

WITHOUT COMMENTS

```
SELECT EmpID, Name, Salary
FROM Employees
WHERE Salary > 50000
AND Department = 'IT';
```

⚠ Hard to understand for others or in future.

WITH COMMENTS

```
-- Get IT department employees
-- who have salary more than 50000
SELECT EmpID, Name, Salary
FROM Employees
WHERE Salary > 50000   -- Salary filter
AND Department = 'IT';-- Department filter
```

✓ Easy to understand, maintain and update.

5) BEST PRACTICES

- Write comments in simple and clear language.
- Explain WHY, not WHAT (code already shows WHAT).
- Keep comments updated with code changes.
- Use comments to explain complex logic.
- Don't over-comment obvious code.

6) USE CASES OF COMMENTS

- Document complex business logic
- Disable code temporarily
- Help teammates understand code
- Add author and date information
- Explain temporary workarounds

7) COMMON MISTAKES

- Forgetting to close /* */ in multi line comments.
- Writing outdated or wrong comments.
- Commenting too much or too little.
- Using comments to hide code instead of version control.
- Not reviewing comments after modifying logic.

8) SUMMARY TABLE

Type	Syntax	Scope	Example	Use
Single Line	--	Till end of line	-- This is a note	Quick notes in a line
Multi Line	/* */	Across multiple lines	/* This is multi line note */	Detailed explanation

★ GOOD CODE IS NOT JUST ABOUT WORKING, IT'S ABOUT BEING UNDERSTANDABLE.

</> USE COMMENTS TO MAKE YOUR SQL CODE CLEAN, SMART & TEAM-FRIENDLY!

SQL AGGREGATE FUNCTIONS

Powerful Functions to Analyze Numeric Data

Aggregate functions perform calculations on a set of values and return a single result. They are widely used in **statistics**, **reports** and **data analysis**.

WHY USE AGGREGATE FUNCTIONS?

- **SUMMARIZE DATA** — Get single value from many rows
- **STATISTICS** — Calculate useful statistics quickly
- **REPORTS** — Create reports and dashboards
- **PERFORMANCE** — Reduce data processing time
- **DECISION MAKING** — Better insights for business decisions

COUNT()

Counts the number of rows or non-null values in a column.

Syntax

```
SELECT COUNT(column_name)
FROM table_name;
```

Example

```
SELECT COUNT(*)
FROM Employees;    -- Total rows
```

SUM()

Returns the total sum of numeric values in a column.

Syntax

```
SELECT SUM(column_name)
FROM table_name;
```

Example

```
SELECT SUM(Salary)
FROM Employees;  -- Total salary
```

AVG()

Returns the average of numeric values in a column.

Syntax

```
SELECT AVG(column_name)
FROM table_name;
```

Example

```
SELECT AVG(Salary)
FROM Employees;  -- Average salary
```

MIN()

Returns the smallest (minimum) value in a column.

Syntax

```
SELECT MIN(column_name)
FROM table_name;
```

Example

```
SELECT MIN(Salary)
FROM Employees;    -- Min salary
```

MAX()

Returns the largest (maximum) value in a column.

Syntax

```
SELECT MAX(column_name)
FROM table_name;
```

Example

```
SELECT MAX(Salary)
FROM Employees;    -- Max salary
```

KEY POINTS

- Works on numeric data except COUNT(*).
- Ignores NULL values (except COUNT(*), COUNT(column)).
- Can be used with WHERE clause.
- Often used with GROUP BY clause.

EXAMPLE TABLE: Employees

EmpID	Name	Department	Salary	Bonus
101	Alice	HR	50000	5000
102	Bob	IT	60000	6000
103	Charlie	Finance	55000	5500
104	Diana	HR	45000	NULL
105	Evan	IT	70000	7000

Total Rows: 5

AGGREGATE FUNCTIONS RESULTS (ON ENTIRE TABLE)

COUNT(*)	COUNT(Salary)	SUM(Salary)	AVG(Salary)	MIN(Salary)	MAX(Salary)	SUM(Bonus)	AVG(Bonus)
5 (Total Rows)	5 (Non-NULL Salaries)	280000 (Total Salary)	56000.00 (Average Salary)	45000 (Minimum Salary)	70000 (Maximum Salary)	23500 (Total Bonus)	5875.00 (Average Bonus)

REPORTS & STATISTICS EXAMPLES

Total number of employees in the company.

```
SELECT COUNT(*) FROM Employees;
```

Total and average salary for budget planning.

```
SELECT SUM(Salary), AVG(Salary) FROM Employees;
```

Find highest and lowest salary for analysis.

```
SELECT MIN(Salary), MAX(Salary) FROM Employees;
```

WITH WHERE CLAUSE

Aggregate on filtered data.

```
SELECT
  SUM(Salary) AS Total_IT_Salary,
  AVG(Salary) AS Avg_IT_Salary
FROM Employees
WHERE Department = 'IT';
```

Result

Total_IT_Salary	Avg_IT_Salary
130000	65000.00

WITH GROUP BY CLAUSE

Aggregate by groups.

```
SELECT Department,
   COUNT(*) AS Total_Employees,
   AVG(Salary) AS Avg_Salary
 FROM Employees
 GROUP BY Department;
```

Result

Department	Total_Employees	Avg_Salary
HR	2	47500.00
IT	2	65000.00
Finance	1	55000.00

COMMON USE CASES

- Counting total records
- Calculating totals and averages
- Finding highest and lowest values
- Generating business reports
- Data analysis and dashboards

IMPORTANT NOTES

- COUNT(*) counts all rows including rows with NULL.
- COUNT(column) counts only non-NULL values in that column.
- Other functions (SUM, AVG, MIN, MAX) ignore NULL values.
- Always use meaningful column names with aliases for clarity.

QUICK SUMMARY

Function	Purpose	Example	Works On
COUNT()	Counts rows/values	COUNT(*)	Any
SUM()	Total of values	SUM(Salary)	Numeric
AVG()	Average of values	AVG(Salary)	Numeric
MIN()	Smallest value	MIN(Salary)	Numeric
MAX()	Largest value	MAX(Salary)	Numeric

Aggregate Functions = Better Insights + Better Decisions | **Use them to turn raw data into meaningful information!** | **Perfect for Statistics, Reports & Data Analysis!**

SQL GROUP BY

Group Rows That Have the Same Values

GROUP BY is used to group rows that have the same values in specified columns and perform aggregate functions on each group.

WHY USE GROUP BY?

- **GROUPING** — Group related data together
- **SUMMARY** — Get summary or totals for each group
- **REPORTS** — Useful in reports & dashboards
- **ANALYSIS** — Analyze data by category/department
- **DECISIONS** — Better insights for business decisions
- **PERFORMANCE** — Improves query readability

KEY CONCEPTS

GROUPING
Rows are grouped based on one or more columns.

AGGREGATION
Use aggregate functions like COUNT(), SUM(), AVG(), MIN(), MAX() with GROUP BY.

SELECT RULE
Columns in SELECT must be in GROUP BY or inside an aggregate function.

FILTER GROUPS
Use HAVING to filter groups after grouping.

ORDER RESULTS
Use ORDER BY to sort the grouped results.

1) SYNTAX

```
SELECT column1, column2, aggregate_function(column3)
FROM table_name
WHERE condition
GROUP BY column1, column2
HAVING condition
ORDER BY column1, column2;
```

AGGREGATE FUNCTIONS

COUNT() | SUM() | AVG() | MIN() | MAX()

2) EXAMPLE DATA

Table: Employees

EmpID	Name	Department	City	Salary
101	Alice	HR	Delhi	50000
102	Bob	HR	Delhi	60000
103	Charlie	IT	Mumbai	55000
104	Diana	IT	Mumbai	45000
105	Evan	Finance	Kolkata	70000
106	Fiza	Finance	Kolkata	65000
107	Grace	HR	Mumbai	40000

3) GROUP BY EXAMPLES

A) Total Employees in Each Department

```
SELECT Department, COUNT(*) AS Total_Employees
FROM Employees
GROUP BY Department;
```

Department	Total_Employees
HR	3
IT	2
Finance	2

B) Total Salary by Department

```
SELECT Department, SUM(Salary) AS Total_Salary
FROM Employees
GROUP BY Department;
```

Department	Total_Salary
HR	150000
IT	100000
Finance	135000

C) Average Salary by City

```
SELECT City, AVG(Salary) AS Avg_Salary
FROM Employees
GROUP BY City;
```

City	Avg_Salary
Delhi	55000.00
Mumbai	46666.67
Kolkata	67500.00

D) Total Salary by Department (More than 100000)

```
SELECT Department, SUM(Salary) AS Total_Salary
FROM Employees
GROUP BY Department
HAVING SUM(Salary) > 100000;
```

Department	Total_Salary
HR	150000
Finance	135000

4) GROUP BY WITH MULTIPLE COLUMNS

Total Employees in Each Department and City

```
SELECT Department, City, COUNT(*) AS Total_Employees
FROM Employees
GROUP BY Department, City
ORDER BY Department, City;
```

Department	City	Total_Employees
Finance	Kolkata	2
HR	Delhi	2
HR	Mumbai	1
IT	Mumbai	2

GROUP BY supports grouping on one or more columns.

5) GROUP BY CATEGORIES (EXAMPLES)

Category	Group products by category and get total sales.
Department	Group employees by department and get headcount or total salary.
City	Group customers or sales by city.
State	Group data by state for regional reports.

6) REAL WORLD BUSINESS USE CASES

Sales Reports	Total sales by product category, region, city.
HR Reports	Employee count and total salary by department.
Finance Reports	Total expenses/income by category, month, state.
Customer Analysis	Customers by city/state, total orders per city.

7) IMPORTANT NOTES

- Every column in SELECT must be in GROUP BY or used inside an aggregate function.
- Use HAVING to filter groups after aggregation.
- WHERE filters rows before grouping.
- GROUP BY works with ORDER BY for sorted results.
- Aliases (AS) make reports more readable.

8) QUICK SUMMARY

Clause	Purpose	Used For
GROUP BY	Groups rows by one or more columns	Creating groups
WITH AGGREGATES	Calculate summary values	COUNT, SUM, AVG, MIN, MAX
HAVING	Filter groups after aggregation	Conditions on aggregated data
ORDER BY	Sort the grouped results	Better readability in reports

TIPS

- Always use meaningful column names.
- Use aggregate functions to get insights.
- Test queries step by step for accuracy.

GROUP BY = GROUPING + SUMMARY + BETTER REPORTS

Group Data + Calculate Summary = Better Reports

COMMON MISTAKES

- Using non-aggregated columns not in GROUP BY.
- Forgetting HAVING for group level filtering.
- Confusing WHERE and HAVING.

REMEMBER

GROUP BY turns raw data into meaningful summaries that help in better reporting and decision making.

SQL HAVING CLAUSE

Filter Groups After Aggregation

HAVING clause is used with GROUP BY to filter groups based on aggregate functions like **COUNT()**, **SUM()**, **AVG()**, **MIN()**, **MAX()** etc.

WHY USE HAVING?

- **FILTER GROUPS** — Filter groups after aggregation
- **AGGREGATE** — Works with aggregate functions
- **SUMMARY** — Get meaningful summaries
- **REPORTS** — Useful for reports and dashboards
- **ACCURATE RESULTS** — Show only groups that meet conditions

KEY CONCEPTS

GROUP BY
Groups rows that have same values in specified columns.

HAVING
Filters the groups created by GROUP BY using aggregate conditions.

AGGREGATE FUNCTIONS
COUNT(), SUM(), AVG(), MIN(), MAX() calculate values for each group.

FILTER GROUPS
HAVING is applied after aggregation, WHERE is applied before grouping.

RESULTS
Returns only those groups that satisfy the HAVING condition.

1) SYNTAX

```
SELECT column1, aggregate_function(column2)
FROM table_name
WHERE condition    -- (optional)
GROUP BY column1, column2    -- grouping
HAVING aggregate_condition;  -- filter groups
```

COMMON AGGREGATE FUNCTIONS

COUNT()	SUM()	AVG()	MIN()	MAX()
Counts rows	Total sum	Average	Minimum	Maximum

2) EXAMPLE DATA (Employees)

EmpID	Name	Department	City	Salary
101	Alice	HR	Delhi	50000
102	Bob	HR	Delhi	60000
103	Charlie	IT	Mumbai	55000
104	Diana	IT	Mumbai	45000
105	Evan	Finance	Kolkata	70000
106	Fiza	Finance	Kolkata	65000
107	Grace	HR	Mumbai	40000
108	Henry	IT	Delhi	75000

We will group data and then filter groups using HAVING.

3) EXAMPLES WITH GROUP BY & HAVING

A) Departments having more than 2 employees

```
SELECT Department, COUNT(*) AS Total_Employees
FROM Employees
GROUP BY Department
HAVING COUNT(*) > 2;
```

Department	Total_Employees
HR	3
IT	4
Finance	2

B) Departments with total salary more than 100000

```
SELECT Department, SUM(Salary) AS Total_Salary
FROM Employees
GROUP BY Department
HAVING SUM(Salary) > 100000;
```

Department	Total_Salary
HR	150000
IT	175000
Finance	135000

C) Cities with average salary greater than 55000

```
SELECT City, AVG(Salary) AS Avg_Salary
FROM Employees
GROUP BY City
HAVING AVG(Salary) > 55000;
```

City	Avg_Salary
Delhi	61666.67
Kolkata	67500.00

4) HAVING vs WHERE

WHERE (Filters Rows Before Grouping)

```
SELECT Department, COUNT(*) AS Emp_Count
FROM Employees
WHERE Salary > 50000
GROUP BY Department;
```

Filters individual rows before grouping.

HAVING (Filters Groups After Grouping)

```
SELECT Department, COUNT(*) AS Emp_Count
FROM Employees
GROUP BY Department
HAVING COUNT(*) > 2;
```

Filters groups after aggregation.

Use WHERE for row-level filters,
Use HAVING for group-level filters.

5) MORE EXAMPLES

D) Departments with average salary >= 55000

```
SELECT Department, AVG(Salary) AS Avg_Salary
FROM Employees
GROUP BY Department
HAVING AVG(Salary) >= 55000;
```

Department	Avg_Salary
HR	50000
IT	58333.33
Finance	67500

E) Departments with total salary between 120000 and 200000

```
SELECT Department, SUM(Salary) AS Total_Salary
FROM Employees
GROUP BY Department
HAVING SUM(Salary) BETWEEN 120000 AND 200000;
```

Department	Total_Salary
HR	150000
IT	175000
Finance	135000

6) GROUP BY MULTIPLE COLUMNS

Total Salary by Department and City (only groups with total > 90000)

```
SELECT Department, City, SUM(Salary) AS TotM Employees
GROUP BY Department, City
HAVING SUM(Salary) > 90000;
```

Department	City	Total_Salary
HR	Delhi	110000
IT	Mumbai	100000
Finance	Kolkata	135000
IT	Delhi	75000 (not shown)

Groups having total salary > 90000 are shown.

7) ORDERING RESULTS

You can use ORDER BY with GROUP BY and HAVING.

```
SELECT Department, COUNT(*) AS Emp_Count
FROM Employees
GROUP BY Department
HAVING COUNT(*) > 1
ORDER BY Emp_Count DESC;
```

Department	Emp_Count
IT	4
HR	3
Finance	2

8) REAL WORLD USE CASES

- **HR Reports** — Departments with more than 5 employees.
- **Finance Reports** — Departments with total salary > budget.
- **Sales Reports** — Products/Category with total sales > target.
- **Regional Reports** — Cities/States with average salary > value.

9) QUICK SUMMARY

Term	Description
GROUP BY	Groups rows with same values.
HAVING	Filters groups using aggregate conditions.
COUNT()	Counts the number of rows in each group.
SUM()	Calculates total for each group.
AVG()	Calculates average for each group.
Filter Groups	Use HAVING after GROUP BY.
Results	Returns only groups that match condition.

KEY FORMULA

GROUP BY + AGGREGATE + HAVING = MEANINGFUL INSIGHTS

REMEMBER

- WHERE filters rows before grouping.
- HAVING filters groups after aggregation.

PRO TIP

- Always use aggregate functions in HAVING clause.
- Use clear aliases for better readability.

BEST PRACTICE

Use GROUP BY to summarize data and HAVING to show only important groups in reports.

SQL LIKE OPERATOR

Pattern Matching & Search Made Easy

LIKE operator is used in WHERE clause to search for a specified pattern in a column.
It supports wildcards % and _ for pattern matching.

WILDCARDS IN LIKE OPERATOR

%	_
PERCENT	**UNDERSCORE**
Represents zero or more characters. Matches any sequence of characters.	Represents a single character. Matches any single character.

COMMON USES

STARTS WITH	ENDS WITH	CONTAINS	PATTERN MATCH	SEARCH
Find values that start with a specific text.	Find values that end with a specific text.	Find values that contain a specific text.	Match a pattern using % and _ wildcards.	Perform flexible search in text data.

KEY POINTS

- **LIKE** is case-insensitive in most databases (depends on collation).
- % matches any number of characters (including none).
- _ matches exactly one character.
- Used in **WHERE** clause for pattern matching.
- Works with text (CHAR, VARCHAR, TEXT, etc.).
- Combine % and _ to create powerful patterns.

EXAMPLE TABLE: Employees

EmpID	Name	Department	City	Email
101	Alice Johnson	HR	Delhi	alice.johnson@company.com
102	Bob Smith	IT	Mumbai	bob.smith@company.com
103	Charlie Brown	IT	Bangalore	charlie.brown@company.com
104	David Wilson	HR	Delhi	david.wilson@company.com
105	Eva Green	Finance	Pune	eva.green@company.com
106	Frank Harris	IT	Hyderabad	frank.harris@company.com
107	Grace Lee	Finance	Mumbai	grace.lee@company.com
108	Henry Clark	HR	Chennai	henry.clark@company.com

EXAMPLES BY SEARCH TYPE

A→	Starts With (A%)	`SELECT * FROM Employees WHERE Name LIKE 'A%';`	Alice Johnson
→Z	Ends With (%son)	`SELECT * FROM Employees WHERE Name LIKE '%son';`	Alice Johnson
A...Z	Contains (%li%)	`SELECT * FROM Employees WHERE Name LIKE '%li%';`	Alice Johnson, Charlie Brown, David Wilson
	Pattern Match (_ob%)	`SELECT * FROM Employees WHERE Name LIKE '_ob%';`	Bob Smith
	Search (Advanced)	`SELECT * FROM Employees WHERE Name LIKE '%an%';`	Frank Harris

LIKE PATTERNS & EXAMPLES

Pattern	Meaning	Example Query	Matches (from table)
`'A%'`	Starts with 'A' (any characters after)	`SELECT * FROM Employees WHERE Name LIKE 'A%';`	Alice Johnson
`'%son'`	Ends with 'son' (any characters before)	`SELECT * FROM Employees WHERE Name LIKE '%son';`	Alice Johnson
`'%li%'`	Contains 'li' anywhere (any characters before/after)	`SELECT * FROM Employees WHERE Name LIKE '%li%';`	Alice Johnson, Charlie Brown, David Wilson
`'_ob%'`	Second letter 'o' and starts with 'ob'	`SELECT * FROM Employees WHERE Name LIKE '_ob%';`	Bob Smith
`'__a%'`	Third letter 'a' (at least 3 letters)	`SELECT * FROM Employees WHERE Name LIKE '__a%';`	Eva Green, Grace Lee
`'%an%'`	Contains 'an' anywhere	`SELECT * FROM Employees WHERE Name LIKE '%an%';`	Frank Harris
`'___y%'`	Starts with 3 letters and 4th letter 'y'	`SELECT * FROM Employees WHERE Name LIKE '___y%';`	Henry Clark
`'%_ee'`	Ends with 'ee' and has at least one character before 'ee'	`SELECT * FROM Employees WHERE Name LIKE '%_ee';`	Eva Green, Grace Lee

MORE PATTERN EXAMPLES

Pattern	Meaning	Example	Matches
`'%IT%'`	Department contains 'IT'	`WHERE Department LIKE '%IT%';`	IT
`'D___%'`	Name starts with 'D' and has at least 4 letters	`WHERE Name LIKE 'D___%';`	David Wilson
`'%@company.com'`	Email ends with '@company.com'	`WHERE Email LIKE '%@company.com';`	All rows
`'___e'`	Name has 4 letters and ends with 'e'	`WHERE Name LIKE '___e';`	Grace Lee

LIKE vs = (EQUAL)

= (EQUAL)	LIKE
Checks exact match.	Checks pattern match.
No wildcards allowed.	Supports % and _ wildcards.
Example: `WHERE Name = 'Alice';`	Example: `WHERE Name LIKE 'A%';`

Use LIKE for flexible search and = for exact match.

TIPS & BEST PRACTICES

- Use % for unknown number of characters.
- Use _ for single character matching.
- LIKE is not case-sensitive in most databases (depends on collation).
- Avoid leading wildcard (%) for better performance (e.g., '%text' is slower).
- Combine with other conditions using AND, OR.

```
SELECT * FROM Employees
WHERE Department LIKE 'IT%'
AND City LIKE 'M%';
```

QUICK SUMMARY

Concept	Description
LIKE	Pattern matching operator.
%	Matches zero or more characters.
_	Matches exactly one character.
Starts With	'text%'
Ends With	'%text'
Contains	'%text%'
Used In	WHERE clause
Purpose	Search & filter text data

LIKE Operator = Powerful Search + Flexible Patterns = Better Results!

Master LIKE, Master Search!

Search smarter. Get better insights.

SQL IN OPERATOR

Filter Data Using Multiple Values

The IN operator is used in WHERE clause to filter records that match **any value** in a list.

WHAT IS IN OPERATOR?

IN operator allows you to specify multiple values in a list and checks if a column matches any of those values.

```
column_name IN (value1, value2, value3, ...)
```

KEY BENEFITS

- **MULTIPLE VALUES** — Check against multiple values at once.
- **FILTER** — Easily filter records based on a list of values.
- **OR ALTERNATIVE** — Simpler and cleaner than using multiple OR conditions.
- **BETTER READABILITY** — Makes queries easy to read and maintain.
- **BETTER PERFORMANCE** — Usually faster than multiple OR conditions.

KEY POINTS

- IN is used with WHERE clause.
- It compares a column with multiple values.
- Syntax: `column_name IN (value1, value2, value3, ...);`
- Values can be numbers, strings, dates, etc.
- If the list is empty, use carefully to avoid errors.
- Acts as an OR alternative: `column = value1 OR column = value2 OR ...`

EXAMPLE TABLE: Employees

EmpID	Name	Department	City	Salary
101	Alice Johnson	HR	Delhi	50000
102	Bob Smith	IT	Mumbai	60000
103	Charlie Brown	IT	Bangalore	55000
104	David Wilson	HR	Delhi	45000
105	Eva Green	Finance	Pune	70000
106	Frank Harris	IT	Hyderabad	65000
107	Grace Lee	Finance	Mumbai	60000
108	Henry Clark	HR	Chennai	50000

IN vs OR (OR ALTERNATIVE)

Using IN (Better)

```
SELECT * FROM Employees
WHERE Department IN ('HR','IT');
```

- Shorter and cleaner
- Easier to read and maintain

Using OR (Longer)

```
SELECT * FROM Employees
WHERE Department = 'HR'
   OR Department = 'IT';
```

- Longer query
- Harder to maintain

EXAMPLES USING IN OPERATOR

Use Case	Query	Description	Result (From Table)
Filter by Multiple Departments	`SELECT * FROM Employees WHERE Department IN ('HR', 'IT');`	Get employees from HR or IT department.	101, 102, 103, 104, 106, 108
Filter by Multiple Cities	`SELECT * FROM Employees WHERE City IN ('Mumbai', 'Delhi');`	Get employees from Mumbai or Delhi.	101, 102, 104, 107
Filter by Salary Range (List)	`SELECT * FROM Employees WHERE Salary IN (50000, 60000, 70000);`	Get employees with salary 50000, 60000 or 70000.	101, 102, 105, 107, 108
Filter by Single Value	`SELECT * FROM Employees WHERE Department IN ('Finance');`	IN also works with a single value.	105, 107
Exclude Values (NOT IN)	`SELECT * FROM Employees WHERE Department NOT IN ('HR');`	Get employees whose department is not HR.	102, 103, 105, 106, 107

WORKS WITH DIFFERENT DATA TYPES

Numbers	Strings	Dates	Mixed List
`SELECT * FROM Employees WHERE EmpID IN (101, 103, 105);`	`SELECT * FROM Employees WHERE Name IN ('Alice Johnson', 'Eva Green');`	`SELECT * FROM Employees WHERE JoinDate IN ('2024-01-01','2024-03-15');`	`SELECT * FROM Employees WHERE City IN ('Delhi','Pune', 'Chennai');`
Result: 101, 103, 105	Result: 101, 105	Result: (Matching dates)	Result: 101, 104, 105, 108

IN OPERATOR WITH SUBQUERY

```
SELECT * FROM Employees
WHERE Department IN (SELECT Department FROM Departments
      WHERE Location = 'Delhi');
```

Returns employees whose department is located in Delhi.

IMPORTANT NOTES

- IN is case-sensitive for text values (depends on collation).
- Use NOT IN to exclude multiple values.
- Avoid large lists; for huge lists, use a temporary table or subquery.
- NULL values are not matched using IN. Use IS NULL for NULL checks.

QUICK SUMMARY

- **IN** — Check column against a list of values
- Filters records that match any value in the list
- Acts as OR Alternative
- Improves query readability and maintainability
- Helps in precise and efficient data selection

SYNTAX

```
SELECT column1, column2, ...
FROM  table_name
WHERE column_name IN (value1, value2, value3, ...);
```

NOT IN to exclude:

```
WHERE column_name NOT IN (value1, value2, ...);
```

BEST PRACTICES

- Use IN for small to medium lists.
- Use subquery for dynamic lists.
- Keep queries simple and readable.
- Always test with different data sets.

IN Operator = Simple, Powerful & Efficient Data Selection! | Use IN for multiple values, simplify your queries, and get better results! | Better Queries, Better Performance, Better Results!

SQL BETWEEN OPERATOR

Filter Data Within a Range (Inclusive)

BETWEEN operator is used in WHERE clause to filter records within a given range.
It works with numbers, dates, and text (alphabetical).

WHY USE BETWEEN?

- **DATE RANGE** — Filter records within a date range.
- **NUMBER RANGE** — Filter records within a numeric range.
- **INCLUSIVE** — Includes both the start and end values.
- **FILTER** — Easily filter records in a range.
- **SEARCH** — Powerful search for specific ranges.
- **QUERY** — Simplify queries compared to multiple conditions.
- **RECORDS** — Get exact set of records you need.

KEY POINTS

- BETWEEN is used in WHERE clause.
- It is INCLUSIVE: Both start and end values are included.
- Works with numbers, dates, and text.
- Good for filtering ranges of data.
- Syntax: `column_name BETWEEN value1 AND value2`
- If value1 is greater than value2, result will be empty.

EXAMPLE TABLE: Employees

EmpID	Name	Department	Salary	JoinDate
101	Alice Johnson	HR	45000	2023-01-15
102	Bob Smith	IT	60000	2022-06-10
103	Charlie Brown	IT	55000	2021-12-05
104	David Wilson	HR	48000	2023-03-20
105	Eva Green	Finance	70000	2022-11-18
106	Frank Harris	IT	65000	2023-02-01
107	Grace Lee	Finance	60000	2021-08-25
108	Henry Clark	HR	50000	2022-09-12

BETWEEN vs OTHER OPERATORS

Using BETWEEN (Better)

```
SELECT * FROM Employees
WHERE Salary BETWEEN 50000 AND 60000;
```

- Easy to write
- Easy to read
- Better performance
- Less typing

Using AND (Longer)

```
SELECT * FROM Employees
WHERE Salary >= 50000
  AND Salary <= 60000;
```

- Longer query
- Harder to read
- More typing

EXAMPLES USING BETWEEN OPERATOR

Use Case	Query	Description	Result (From Example Table)
Date Range	`SELECT * FROM Employees WHERE JoinDate BETWEEN '2022-01-01' AND '2022-12-31';`	Get employees who joined in year 2022.	102, 105, 108
Number Range	`SELECT * FROM Employees WHERE Salary BETWEEN 50000 AND 60000;`	Get employees with salary between 50000 and 60000.	102, 103, 107, 108
Inclusive (Includes Start and End)	`SELECT * FROM Employees WHERE Salary BETWEEN 50000 AND 70000;`	Includes employees with salary exactly 50000 and 70000.	101, 102, 103, 105, 107, 108
Filter by Department and Range	`SELECT * FROM Employees WHERE Department = 'IT' AND Salary BETWEEN 55000 AND 65000;`	Get IT department employees with salary between 55000 and 65000.	102, 103, 106
Text (Alphabetical)	`SELECT * FROM Employees WHERE Name BETWEEN 'C' AND 'H';`	Get employees whose name is between C and H (alphabetically).	103, 104, 105, 106, 107
NOT BETWEEN	`SELECT * FROM Employees WHERE Salary NOT BETWEEN 50000 AND 60000;`	Get employees whose salary is not between 50000 and 60000.	101, 104, 105, 106

BETWEEN WITH DIFFERENT DATA TYPES

NUMBERS

```
SELECT * FROM Employees
WHERE Salary BETWEEN 45000 AND 65000;
```

Filters numeric values between 45000 and 65000.

Result: 101, 102, 103, 104, 106, 107, 108

DATES

```
SELECT * FROM Employees
WHERE JoinDate BETWEEN '2022-01-01'
AND '2023-01-01';
```

Filters dates between 1 Jan 2022 and 1 Jan 2023.

Result: 102, 105, 108

TEXT (ALPHABETICAL)

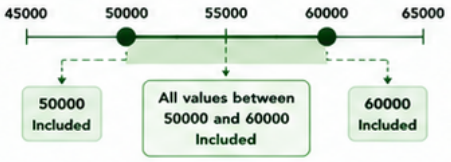
```
SELECT * FROM Employees
WHERE Name BETWEEN 'D' AND 'G';
```

Filters text alphabetically from D to G.

Result: 104, 105, 106, 107

BETWEEN IS INCLUSIVE

Example: Salary BETWEEN 50000 AND 60000



Both the start value (50000) and end value (60000) are included.

BETWEEN WITH NULL VALUES

NULL values are NOT returned in BETWEEN.

```
SELECT * FROM Employees
WHERE Salary BETWEEN 50000 AND 60000;
```

Rows where Salary is NULL will not be shown.

Use IS NULL / IS NOT NULL to handle NULL values.

QUICK SYNTAX

```
-- Basic Syntax
SELECT column1, column2, ...
FROM table_name
WHERE column_name
BETWEEN value1 AND value2;

-- Example
SELECT * FROM Employees
WHERE Salary BETWEEN 50000 AND 60000;
```

BEST PRACTICES

- Use BETWEEN for range filtering to make queries simple and readable.
- Ensure the first value is smaller than or equal to the second value.
- Use proper date format (YYYY-MM-DD) for date ranges.
- Index columns (like Date, Salary) for better performance.

COMMON MISTAKES

- Using BETWEEN for values that are not in sequence.
- Forgetting that BETWEEN is INCLUSIVE.
- Using text ranges without understanding alphabetical order.

SUMMARY

- BETWEEN helps to filter records within a range.
- It works with numbers, dates, and text.
- It is INCLUSIVE (includes start and end values).
- It is easier to read and write compared to using AND conditions.

BETWEEN = Better Range Filtering!

SQL IS NULL / IS NOT NULL

Find Missing or Available Data Easily

IS NULL and IS NOT NULL operators are used in WHERE clause to check for missing (NULL) or available (not NULL) values in a column.

WHY USE IS NULL / IS NOT NULL?

- **MISSING DATA** — Find records with missing or empty values (NULL).
- **DATA QUALITY** — Identify incomplete records and clean your data.
- **FILTERING** — Filter records based on NULL or NOT NULL values.
- **SEARCH** — Search for unknown or available information.
- **REPORTS** — Create accurate reports with valid data only.
- **VALIDATION** — Validate required fields and improve data integrity.

KEY POINTS

- NULL represents missing, unknown or not applicable data.
- Use IS NULL to find records where column value is NULL.
- Use IS NOT NULL to find records where column has a value.
- NULL is not equal to 0, empty string ('') or space.
- Use these operators in WHERE clause only.

NULL vs EMPTY STRING

NULL	Empty String ('')
• Means no value	• Means empty value
• Unknown / Missing	• Still a value
• No space stored	• Space is stored
• Use IS NULL	• Use = '' to check

NULL is NOT same as empty string or 0.

EXAMPLE TABLE: Employees

EmpID	Name	Department	Email	Phone	Salary
101	Alice Johnson	HR	alice@company.com	9876543210	45000
102	Bob Smith	IT	bob@company.com	NULL	60000
103	Charlie Brown	IT	NULL	9123456780	55000
104	David Wilson	HR	david@company.com	NULL	NULL
105	Eva Green	Finance	eva@company.com	9988776655	70000
106	Frank Harris	IT	NULL	NULL	65000
107	Grace Lee	Finance	grace@company.com	9001122334	60000
108	Henry Clark	HR	NULL	8765432101	50000

NULL CHECK COMBINATIONS

Email or Phone is NULL	`SELECT * FROM Employees WHERE Email IS NULL OR Phone IS NULL;`	Email or Phone is missing.	102, 103, 104, 106, 108
Email and Phone are NULL	`SELECT * FROM Employees WHERE Email IS NULL AND Phone IS NULL;`	Both Email and Phone are missing.	106
Either Salary is NULL or < 50000	`SELECT * FROM Employees WHERE Salary IS NULL OR Salary < 50000;`	Salary missing or less than 50000.	101, 104, 108

EXAMPLES USING IS NULL / IS NOT NULL

Use Case	Query	Description	Result (From Example Table)
Find employees with no Email	`SELECT * FROM Employees WHERE Email IS NULL;`	Get employees whose Email is missing.	103, 106, 108
Find employees with Email	`SELECT * FROM Employees WHERE Email IS NOT NULL;`	Get employees whose Email is available.	101, 102, 104, 105, 107
Find employees with no Phone	`SELECT * FROM Employees WHERE Phone IS NULL;`	Get employees whose Phone is missing.	102, 104, 106
Find employees with Phone	`SELECT * FROM Employees WHERE Phone IS NOT NULL;`	Get employees whose Phone is available.	101, 103, 105, 107, 108
Find employees with no Salary	`SELECT * FROM Employees WHERE Salary IS NULL;`	Get employees whose Salary is missing.	104
Find employees with Salary	`SELECT * FROM Employees WHERE Salary IS NOT NULL;`	Get employees whose Salary is available.	101, 102, 103, 105, 106, 107, 108

WHERE vs NULL / NOT NULL

INCORRECT (Don't use =)

```
-- This will NOT work for NULL
SELECT * FROM Employees
WHERE Email = NULL;      -- Wrong
```

NULL cannot be compared using =, >, <, etc.

CORRECT (Use IS NULL)

```
-- This is the correct way
SELECT * FROM Employees
WHERE Email IS NULL;     -- Right
```

Always use IS NULL and IS NOT NULL.

NULL IN CONDITIONS

Logical Operations with NULL

```
NULL = NULL          -- Unknown (Not True)
NULL <> NULL         -- Unknown (Not True)
NULL > 10            -- Unknown (Not True)
NULL < 10            -- Unknown (Not True)
NULL IS NULL         -- True
NULL IS NOT NULL     -- False
```

Use IS NULL / IS NOT NULL

Always use IS NULL or IS NOT NULL to check for NULL values.

PRACTICAL EXAMPLES

1. Count NULL Emails

```
SELECT COUNT(*) AS NullEmails
FROM Employees
WHERE Email IS NULL;
```

Result: 3

2. Count NOT NULL Phones

```
SELECT COUNT(*) AS HavePhone
FROM Employees
WHERE Phone IS NOT NULL;
```

Result: 5

3. Update NULL Salary

```
UPDATE Employees
SET Salary = 50000
WHERE Salary IS NULL;
```

Result: Salary updated for 104

4. Delete Records with NULL Email

```
DELETE FROM Employees
WHERE Email IS NULL;
```

Result: Records deleted (103, 106, 108)

QUICK SUMMARY

Operator	Use	Purpose	Example
IS NULL	WHERE column IS NULL	Find missing values	Email IS NULL
IS NOT NULL	WHERE column IS NOT NULL	Find available values	Phone IS NOT NULL

BEST PRACTICES

- Use IS NULL / IS NOT NULL for accurate NULL checks.
- Validate important fields and handle missing data.
- Clean NULL data regularly for better performance.
- Use NULL checks in reports and dashboards.

COMMON MISTAKES

- Using = NULL or <> NULL. (Always use IS NULL / IS NOT NULL)
- Thinking NULL is same as empty string or 0.
- Ignoring NULL checks in important columns.

TIP

Use COALESCE() or ISNULL() functions to replace NULL with a default value.
Example: COALESCE(Email, 'No Email')

REMEMBER

NULL means unknown or missing.
Use IS NULL / IS NOT NULL for correct results and better data quality.

SQL CASE STATEMENT

Add Conditional Logic in Your Queries

CASE statement is used to perform IF...ELSE logic inside SQL queries and return customized output.

CASE KEYWORDS

CASE	WHEN	THEN	ELSE	END	OUTPUT	REPORTS
Starts the CASE expression.	Specifies a condition to evaluate.	Returns a result when condition is true.	Returns a result when no condition is true.	Ends the CASE expression.	Produces custom output based on conditions.	Helpful in reports, dashboards and data analysis.

KEY POINTS

- CASE works like IF...ELSE in other programming languages.
- It can be used in SELECT, WHERE, ORDER BY, GROUP BY, HAVING.
- Makes reports more readable and meaningful.
- Helps in data categorization, bucketing and labeling.
- Evaluate WHEN conditions in order — first match will be executed.

EXAMPLE TABLE: Employees

EmpID	Name	Department	Salary	JoinDate
101	Alice Johnson	HR	45000	2022-01-15
102	Bob Smith	IT	60000	2021-06-10
103	Charlie Brown	IT	85000	2020-03-05
104	David Wilson	HR	48000	2023-02-18
105	Eva Green	Finance	70000	2021-11-20
106	Frank Harris	IT	65000	2022-07-30
107	Grace Lee	Finance	55000	2023-01-12
108	Henry Clark	HR	40000	2022-10-05

SYNTAX: SIMPLE CASE

```
CASE expression
  WHEN value1 THEN result1
  WHEN value2 THEN result2
  ...
  ELSE result_else
END AS alias_name
```

Compares expression with multiple values.

SYNTAX: SEARCHED CASE

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  ...
  ELSE result_else
END AS alias_name
```

Evaluates conditions and returns first matched result.

CASE EXAMPLES IN SELECT

Use Case	Query Example	Description	Output (From Example Table)
Basic CASE (Categorize by Salary)	`SELECT Name, Salary, CASE WHEN Salary >= 70000 THEN 'High' WHEN Salary >= 50000 THEN 'Medium' ELSE 'Low' END AS SalaryLevel FROM Employees;`	Categorizes employees into High, Medium, or Low salary.	Alice-Low, Bob-Medium, Charlie-High, David-Low, Eva-High, Frank-Medium, Grace-Medium, Henry-Low
Multiple Conditions (Department Label)	`SELECT Name, Department, CASE Department WHEN 'HR' THEN 'Human Resources' WHEN 'IT' THEN 'Information Tech' WHEN 'Finance' THEN 'Finance Dept' ELSE 'Other Dept' END AS DeptLabel FROM Employees;`	Returns custom department labels.	HR → Human Resources IT → Information Tech Finance → Finance Dept
With ELSE (Default Value)	`SELECT Name, Salary, CASE WHEN Salary >= 70000 THEN 'High' WHEN Salary >= 50000 THEN 'Medium' ELSE 'Below 50000' END AS SalaryLevel FROM Employees;`	Uses ELSE to handle unmatched conditions.	Henry → Below 50000
With Date (Experience Bucket)	`SELECT Name, JoinDate, CASE WHEN JoinDate >= '2023-01-01' THEN 'New' WHEN JoinDate >= '2021-01-01' THEN 'Experienced' ELSE 'Senior' END AS ExperienceLevel FROM Employees;`	Groups employees by joining date.	New: David, Grace Experienced: Alice, Bob, Eva, Frank, Henry Senior: Charlie
	`SELECT Name, Salary, CASE WHEN Salary BETWEEN 0 AND 49999 THEN '0 - 49K' WHEN Salary BETWEEN 50000 AND 69999 THEN '50K - 69K' WHEN Salary >= 70000 THEN '70K+' END AS SalaryRange FROM Employees;`	Creates custom salary range buckets.	0–49K: Alice, David, Henry 50K–69K: Bob, Frank, Grace 70K+: Charlie, Eva

SIMPLE vs SEARCHED CASE EXAMPLE

SIMPLE CASE Example

```
SELECT Name, Department,
 CASE Department
  WHEN 'HR' THEN 'Human Resources'
  WHEN 'IT' THEN 'Information Tech'
  ELSE 'Finance Dept'
 END AS DeptType
FROM Employees;
```

SEARCHED CASE Example

```
SELECT Name, Salary,
 CASE
  WHEN Salary >= 70000 THEN 'High'
  WHEN Salary >= 50000 THEN 'Medium'
  ELSE 'Low'
 END AS SalaryLevel
FROM Employees;
```

Use SIMPLE CASE when comparing one expression with values.
Use SEARCHED CASE when using conditions.

USING CASE IN WHERE (Advanced Filtering)

Example: Get High Salary Employees

```
SELECT * FROM Employees
WHERE CASE
    WHEN Salary >= 70000 THEN 1
    ELSE 0
  END = 1;
```

Works like: WHERE Salary >= 70000

Example: Custom Filter

```
SELECT * FROM Employees
WHERE CASE Department
    WHEN 'IT' THEN 1
    ELSE 0
  END = 1;
```

Returns only IT department employees.

TIPS & BEST PRACTICES

- Use ELSE to handle all other cases.
- Keep conditions in logical order.
- Use CASE for better readability in reports.
- Alias your output for meaningful column names.

COMMON USE CASES

- Salary level classification
- Performance rating
- Status mapping (Active/Inactive)
- Age group or experience bucket
- Custom sorting in reports

NULL HANDLING WITH CASE

```
SELECT Name, Email,
 CASE
  WHEN Email IS NULL THEN 'No Email'
  ELSE Email
 END AS EmailStatus
FROM Employees;
```

Use CASE to replace NULL with meaningful text.

QUICK SUMMARY

Feature	Details
Purpose	Add conditional logic in SQL queries
Main Keywords	CASE, WHEN, THEN, ELSE, END
Returns	A result based on conditions
Used In	SELECT, WHERE, ORDER BY, GROUP BY, HAVING
Helps In	Reports, Categorization, Data Validation

REMEMBER

- CASE = IF...ELSE in SQL
- First TRUE WHEN is executed
- ELSE is optional but recommended
- END closes the CASE statement
- Great for smart reports!

SQL ALIASES

Give Your Columns and Tables Short, Meaningful Names

Aliases are used to give columns or tables temporary names using **AS**.
They make queries easy to read and reports more understandable.

WHY USE ALIASES?

EASY TO READ	SHORT NAME	REPORTS	CLARITY	FOCUS	OUTPUT	MAINTAINABLE
Makes query more readable and clean.	Use short names for long table or column names.	Perfect for reports and dashboards.	Improves clarity and reduces complexity.	Helps focus on important output columns.	Produces clean, professional output.	Easier to maintain and modify queries.

KEY POINTS

- AS keyword is used to create aliases.
- Aliases are temporary names (only for the current query).
- Column aliases rename output column names.
- Table aliases provide short names for tables.
- Aliases improve readability and reporting.
- Aliases do not exist in the database.

EXAMPLE TABLE: Employees

EmpID	FirstName	LastName	Department	Salary	JoinDate
101	Alice	Johnson	HR	45000	2022-01-15
102	Bob	Smith	IT	60000	2021-06-10
103	Charlie	Brown	IT	55000	2020-03-05
104	David	Wilson	HR	48000	2023-02-18
105	Eva	Green	Finance	70000	2021-11-20
106	Frank	Harris	IT	65000	2022-07-30
107	Grace	Lee	Finance	55000	2023-01-12
108	Henry	Clark	HR	40000	2022-10-05

SYNTAX

Column Alias

```
SELECT ColumnName AS AliasName
FROM TableName;

-- OR (AS is optional)
SELECT ColumnName AliasName
FROM TableName;
```

Table Alias

```
SELECT ColumnName(s)
FROM TableName AS AliasName;

-- OR
SELECT ColumnName(s)
FROM TableName AliasName;
```

EXAMPLES OF SQL ALIASES

Type	Query Example	Description	Output (Column Names)

Column Alias	`SELECT FirstName AS Name, Salary AS MonthlySalary FROM Employees;`	Renames columns in the result set.	`Name | MonthlySalary`
			`Alice | 45000`
			`Bob | 60000`
			`Charlie | 55000`
			`... |`

Table Alias	`SELECT e.FirstName, e.Salary FROM Employees AS e;`	Gives a short name (e) to the table Employees.	`FirstName | Salary`
			`Alice | 45000`
			`Bob | 60000`
			`... |`

Both Table & Column Alias	`SELECT e.FirstName AS EmployeeName, e.Salary AS Pay FROM Employees AS e;`	Uses aliases for both table and columns.	`EmployeeName | Pay`
			`Alice | 45000`
			`Bob | 60000`
			`... |`

Expression Alias	`SELECT FirstName + ' ' + LastName AS FullName, Salary * 12 AS AnnualSalary FROM Employees;`	Aliases for calculated columns.	`FullName | AnnualSalary`
			`Alice Johnson | 540000`
			`Bob Smith | 720000`
			`... |`

A-Z Alias with Spaces	`SELECT FirstName AS [First Name], Salary AS "Monthly Salary" FROM Employees;`	Use brackets [] or quotes " " for alias with spaces.	`First Name | Monthly Salary`
			`Alice | 45000`
			`Bob | 60000`
			`... |`

COLUMN ALIAS EXAMPLES

Query	Output Column Names
`SELECT EmpID AS ID, FirstName AS Name, Salary AS Pay FROM Employees;`	`ID | Name | Pay`
`SELECT FirstName, LastName AS Surname, Department AS Dept FROM Employees;`	`FirstName | Surname | Dept`
`SELECT Salary AS "Monthly Salary", Salary*12 AS "Annual Salary" FROM Employees;`	`Monthly Salary | Annual Salary`
`SELECT COUNT(*) AS TotalEmployees FROM Employees;`	`TotalEmployees`

TABLE ALIAS EXAMPLE

Query	Explanation
`SELECT e.FirstName, e.Salary, d.DepartmentName FROM Employees AS e JOIN Departments AS d ON e.DepartmentID = d.DepartmentID;`	`e` is alias for Employees table. `d` is alias for Departments table.

Using aliases makes joins and long queries shorter and easier to write.

ALIAS VS ACTUAL NAME

Without Alias

```
SELECT Employees.FirstName,
Employees.LastName,
Employees.Salary
FROM Employees;
```

With Alias

```
SELECT e.FirstName,
e.LastName,
e.Salary
FROM Employees AS e;
```

Alias version is shorter, cleaner and easier to read.

QUICK SUMMARY

AS Keyword	Used to create alias.
Column Alias	Renames column in output.
Table Alias	Gives short name to table.
Optional	AS keyword is optional.
Scope	Alias exists only in the query.
Benefit	Better readability and reports.

BEST PRACTICES

- Use meaningful and short aliases.
- Use aliases to improve readability.
- Use consistent names in complex queries.
- Avoid using spaces (use underscrores instead).

COMMON MISTAKES

- Forgetting AS in complex queries.
- Using same alias name for different columns.
- Using reserved keywords as alias.
- Creating aliases that are not meaningful.

USE CASES

- Creating clean reports and dashboards.
- Simplifying complex joins.
- Renaming columns for end users.
- Improving query readability and maintenance.

PRO TIP

Good aliases = Clean Queries
→ Easy to understand
→ Easy to maintain
→ Great for reports and presentations!

SQL VIEWS

Simplify Queries, Secure Data, Build Better Reports

A View in SQL is a virtual table based on the result set of a SELECT query. It does not store data physically.
It improves reusability, security and makes queries easier to manage.

KEY BENEFITS OF SQL VIEWS

CREATE VIEW	VIRTUAL TABLE	SELECT	REUSABLE QUERY	SECURITY	REPORTS	READ ONLY	VIEW
Create a virtual table using a SELECT query.	Acts like a table but stores no data physically.	Access data easily using simple SELECT.	Use the same logic multiple times.	Restrict access to sensitive columns and data.	Perfect for reports, dashboards and analytics.	Views are read-only by default (data remains safe).	See data the way you want, not the way it is.

WHAT IS A VIEW?

Base Tables → VIEW (Virtual Table, Result of SELECT)

- A view is a saved SELECT query.
- It shows data from one or more tables.
- It does not store data.
- It behaves like a table.
- Changes in base tables reflect in the view.

EXAMPLE BASE TABLES

Employees Table

EmpID	Name	DeptID	Salary	LocationID
1	Alice	10	60000	1
2	Bob	20	55000	2
3	Charlie	10	70000	1
4	David	30	50000	3
5	Eva	20	62000	2

Departments Table

DeptID	DeptName
10	IT
20	HR
30	Finance

Views pull data from one or more base tables.

CREATE VIEW EXAMPLES

Use Case	SQL Example	Description
Simple View	`CREATE VIEW vw_EmployeeInfo AS SELECT EmpID, Name, Salary, DeptID FROM Employees;`	Creates a view with selected columns from Employees.
Join View	`CREATE VIEW vw_EmployeeDept AS SELECT e.EmpID, e.Name, d.DeptName, e.Salary FROM Employees e JOIN Departments d ON e.DeptID = d.DeptID;`	View with data from multiple tables using JOIN.
Filtered View	`CREATE VIEW vw_HighSalary AS SELECT EmpID, Name, Salary FROM Employees WHERE Salary >= 60000;`	View with a condition (only high salary employees).
Complex View	`CREATE VIEW vw_EmployeeSummary AS SELECT d.DeptName, COUNT(*) AS TotalEmp, AVG (e.Salary) AS AvgSalary FROM Employees e JOIN Departments d ON e.DeptID = d.DeptID GROUP BY d.DeptName;`	View with aggregation (summary by department).

SYNTAX

```
CREATE VIEW view_name AS
 SELECT column1, column2, ...
 FROM table_name
 WHERE condition;
```

No data is stored when you create a view.
It is a virtual (logical) table.

USING VIEWS

Once created, a view can be used like a normal table.

Action	SQL Example
Select from View	`SELECT * FROM vw_EmployeeInfo;`
Filter View	`SELECT Name, Salary FROM vw_HighSalary WHERE Salary > 65000;`
Join with View	`SELECT v.Name, v.Salary, d.DeptName FROM vw_EmployeeInfo v JOIN Departments d ON v.DeptID = d.DeptID;`
Use in Reports	Use the view in reporting tools (Power BI, Excel, etc.).

READ ONLY BY DEFAULT

Views are read-only to protect base data.

Allowed	Not Allowed
SELECT (Read Data)	INSERT UPDATE DELETE (Unless view is updatable)

To make changes, modify the base tables, not the view.

VIEWS FOR REPORTS & ANALYTICS

- Simplify report queries.
- Ensure consistent data logic.
- Reduce errors in reports.
- Save time by reusing views.
- Perfect for dashboards and KPIs.

One view, multiple reports!

VIEW vs TABLE

Feature	Table	View
Storage	Stores data physically	Does not store data
Data Changes	Data can be inserted, updated, deleted	Usually read-only (depends on view)
Complexity	Data may be complex	Simplifies complex data
Security	Shows all data	Shows only allowed data
Performance	Fast for simple queries	Slightly slower (depends on base tables)

SECURITY WITH VIEWS

Views help hide sensitive columns and provide controlled access.

Base Table (EmpID, Name, Salary, Salary, Bonus, SSN) → View (EmpID, Name, DeptName)

Users can see only what the view allows.
Sensitive data (Bonus, SSN) is hidden.

ALTER, DROP & RENAME VIEWS

```
ALTER VIEW view_name AS ...          -- Modify a view
DROP VIEW view_name;                 -- Delete a view
RENAME VIEW old_name TO new_name;    -- Rename a view
```

BEST PRACTICES

- Use meaningful names for views.
- Keep views simple and focused.
- Test performance for complex views.
- Use views for security, not as a replacement for proper permissions.

COMMON USE CASES

- Simplify complex joins and queries.
- Restrict access to sensitive data.
- Create consistent data for reports.
- Provide a logical layer over the database.

QUICK SUMMARY

Views are virtual tables based on SELECT queries.
They improve reusability, security and reporting.
They are easy to use and make queries clean, simple and powerful.

SQL INDEXES

Make Your Queries Faster and More Efficient!

An INDEX is a special lookup structure that improves the speed of data retrieval operations on a table.
Indexes reduce the time taken for searches and sorting.

WHY USE INDEXES?

- **PERFORMANCE** — Speeds up SELECT queries and reduces response time.
- **FAST SEARCH** — Find data quickly without scanning the whole table.
- **OPTIMIZATION** — Helps the query optimizer choose the best plan.
- **SPEED** — Improves overall database speed and efficiency.
- **UNIQUE & INTEGRITY** — UNIQUE INDEX ensures no duplicate values.
- **BETTER REPORTS** — Faster lookups mean better performance for reports.

KEY POINTS

- An index is like the index page of a book.
- Indexes are used by the WHERE, JOIN, ORDER BY, GROUP BY clauses.
- Indexes improve read performance (SELECT).
- Too many indexes can slow down INSERT, UPDATE and DELETE.
- Choose columns with high cardinality (unique values) for better results.

EXAMPLE TABLE: Employees

EmpID (PK)	Name	Department	Salary	Email
101	Alice Johnson	HR	55000	alice@company.com
102	Bob Smith	IT	60000	bob@company.com
103	Charlie Brown	IT	70000	charlie@company.com
104	David Wilson	Finance	65000	david@company.com
105	Eva Green	HR	54000	eva@company.com
106	Frank Harris	IT	62000	frank@company.com
107	Grace Lee	Finance	58000	grace@company.com
108	Henry Clark	HR	53000	henry@company.com

Without index → Database scans all rows (Slow)
With index → Database finds data quickly (Fast)

CREATE INDEX

CREATE INDEX is used to create an index on a table.

```
-- Syntax
CREATE INDEX index_name
ON table_name (column_name);
```

Examples

```
-- Index on single column
CREATE INDEX idx_emp_name
ON Employees (Name);

-- Index on multiple columns
CREATE INDEX idx_dept_salary
ON Employees (Department, Salary);
```

BENEFITS OF INDEXES

- **FAST SEARCH & LOOKUP** — Indexes act like shortcuts to find data without scanning every row.
- **BETTER QUERY PERFORMANCE** — Reduces I/O operations and speeds up complex queries.
- **OPTIMIZED JOINS & SORTING** — Speeds up JOIN, ORDER BY, GROUP BY and DISTINCT queries.
- **IMPROVED REPORTS** — Reports and dashboards run faster with proper indexing.
- **OVERALL OPTIMIZATION** — Helps the database engine choose the best execution plan.

UNIQUE INDEX

Ensures all values in the indexed column(s) are unique.

```
-- Syntax
CREATE UNIQUE INDEX index_name
ON table_name (column_name);
```

Example

```
CREATE UNIQUE INDEX idx_email
ON Employees (Email);

-- Ensures Email must be unique
```

HOW INDEX WORKS?

Without Index (Full Table Scan)

- Database reads every row one by one.
- Slow for large tables.

With Index (Indexed Search)

- Index quickly locates the data.
- Only matching rows are read.

USING INDEX IN QUERIES

Indexes are automatically used by the database optimizer.

Example Query

```
SELECT * FROM Employees
WHERE Department = 'IT';
```

With index on Department → Fast lookup
Without index → Full table scan (Slow)

CHECK & DROP INDEX

```
-- View Indexes on a Table
SHOW INDEX FROM Employees;
```

Key_name	Column_name	Non_unique
PRIMARY	EmpID	0
idx_dept	Department	1

```
-- Drop an Index
DROP INDEX idx_dept
ON Employees;
```

WHEN TO USE INDEXES?

- Columns used in WHERE clause frequently.
- Columns used in JOIN conditions.
- Columns used in ORDER BY or GROUP BY.
- Columns with unique or few duplicate values.
- Large tables with millions of rows.

TYPES OF INDEXES

PRIMARY INDEX	Automatically created for PRIMARY KEY.	Unique + Not Null
UNIQUE INDEX	Ensures all values in the column(s) are unique.	Unique values allowed only once
SINGLE COLUMN INDEX	Created on a single column.	Improves search on one column
COMPOSITE INDEX	Created on multiple columns.	Great for multi-column search/filter
CLUSTERED INDEX	Sorts and stores data rows based on index.	One per table (some DBs)
NON-CLUSTERED INDEX	Separate structure from data rows.	Multiple allowed per table

PERFORMANCE COMPARISON

Operation	Without Index (Full Scan)	With Index (Indexed Search)
SELECT with WHERE	Slow	Fast
JOIN Operations	Slow	Fast
ORDER BY / GROUP BY	Slow	Fast
INSERT / UPDATE / DELETE	Faster	Slightly Slower
Storage Space	Less	More (Extra Space)

Indexes improve read speed but can slow down write operations and use extra storage.

REAL LIFE ANALOGY

Think of an index like the index of a book.

Without Index

You read every page to find a topic.

With Index

You check the index and go directly to the page.

Indexes = Fast Lookup = Better Performance

QUICK SUMMARY

- **INDEX** — Speeds up data retrieval.
- **CREATE INDEX** — Used to create an index.
- **UNIQUE INDEX** — Ensures uniqueness of data.
- **FAST SEARCH** — Quick lookup and better performance.
- **OPTIMIZATION** — Helps database optimizer.
- **SPEED** — Improves overall system speed.
- **LOOKUP** — Reduces search time.
- **VIEW & REPORTS** — Faster queries = Better reports.

BEST PRACTICES

- Create indexes on columns that are searched frequently.
- Avoid indexing columns with very low or very high duplication.
- Review and remove unused indexes regularly.
- Do not create too many indexes on a table.
- Monitor index usage and performance.
- Use appropriate index type for the right scenario.

COMMON MISTAKES

- Indexing every column unnecessarily.
- Creating indexes on small tables (not useful).
- Forgetting the impact on INSERT/UPDATE/DELETE.

PRO TIP

The right index on the right column can make your queries 10x to 100x faster!

SQL INNER JOIN

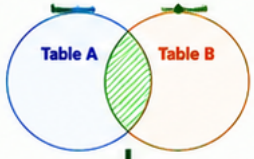
Combine Data from Two Tables – Get Common Records!

INNER JOIN returns only the matching (common) records from two tables based on a related column condition.

KEY CONCEPTS

INNER JOIN	COMMON RECORDS	MATCHING DATA	TWO TABLES	RELATIONSHIP	ON	MERGE	RESULT
Returns only matching rows from both tables.	Shows records that exist in both tables.	Matches related data using a condition.	Works with exactly two tables.	Based on a relationship between columns.	Condition is written using ON keyword.	Combines data from both tables.	Produces the final result set.

HOW INNER JOIN WORKS



INNER JOIN
(Show Only Common)

- Returns only matching rows.
- Removes non-matching rows from both tables.
- Based on the ON condition.

EXAMPLE TABLES

Employees Table

EmpID (PK)	EmpName	DeptID	Salary
1	Alice	10	60000
2	Bob	20	55000
3	Charlie	10	70000
4	David	30	50000
5	Eva	40	65000

Departments Table

DeptID (PK)	DeptName	Location
10	IT	New York
20	HR	London
30	Finance	Delhi
50	Marketing	Sydney

Relation: Employees.DeptID → Departments.DeptID
Common DeptIDs: 10, 20, 30 (DeptID 40 in Employees and 50 in Departments will not appear in result)

SYNTAX

```
SELECT  a.column1, a.column2, b.column1, b.column2, ...
FROM    TableA a
INNER JOIN TableB b
ON      a.common_column = b.common_column;
```

INNER JOIN keyword returns rows when there is at least one match in both tables based on the ON condition.

INNER JOIN EXAMPLE

```
SELECT  e.EmpID, e.EmpName, e.DeptID, e.Salary,
        d.DeptName, d.Location
FROM    Employees e
INNER JOIN Departments d
ON      e.DeptID = d.DeptID;
```

RESULT (INNER JOIN OUTPUT)

EmpID	EmpName	DeptID	Salary	DeptName	Location
1	Alice	10	60000	IT	New York
2	Bob	20	55000	HR	London
3	Charlie	10	70000	IT	New York
4	David	30	50000	Finance	Delhi

Only common records are returned (DeptID 10, 20, 30).
DeptID 40 (Eva) and DeptID 50 (Marketing) are not in both tables.

VISUAL EXPLANATION

Employees

EmpID	EmpName	DeptID
1	Alice	10
2	Bob	20
3	Charlie	10
4	David	30
5	Eva	40

Departments

DeptID	DeptName
10	IT
20	HR
30	Finance
50	Marketing

INNER JOIN RESULT
Only 10, 20, 30 are in both tables.

REAL LIFE ANALOGY

INNER JOIN is like finding employees who belong to departments that exist in both lists.

TYPES OF JOIN (QUICK VIEW)

INNER JOIN	Returns matching rows from both tables.
LEFT JOIN	All rows from left table + matching from right.
RIGHT JOIN	All rows from right table + matching from left.
FULL OUTER JOIN	All rows from both tables (matched + unmatched).

WHEN TO USE INNER JOIN?

- When you need only matching data from both tables.
- When ensuring referential relationship between tables.
- In reports where only valid linked data is required.
- When unwanted or unmatched rows should be excluded.

BEST PRACTICES

- Always use meaningful table aliases.
- Ensure ON condition uses correct related columns.
- Index the join columns for better performance.
- Select only the required columns.
- Test the query with sample data.

PERFORMANCE TIPS

- Create indexes on join columns (e.g., EmpID, DeptID).
- Avoid using functions on join columns.
- Use INNER JOIN instead of WHERE for joining.
- Filter data early using WHERE before joining (if possible).

COMMON MISTAKES

- Forgetting the ON condition (causes cartesian product).
- Joining on wrong or unrelated columns.
- Using SELECT * instead of specific columns.
- Expecting unmatched rows in INNER JOIN.

QUICK SUMMARY

INNER JOIN returns only the common (matching) records between two tables using the ON condition.
It is the most commonly used join for combining related data and building clean, accurate results.

SQL LEFT JOIN

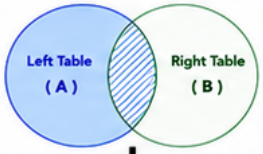
Get All Rows from Left Table and Matching Rows from Right Table

LEFT JOIN returns all rows from the left table, and the matching rows from the right table. If no match is found, NULL values are returned for columns from the right table.

KEY CONCEPTS

LEFT JOIN	ALL LEFT ROWS	MATCHING RIGHT	NULL VALUES	RELATIONSHIP	JOIN	QUERY	RESULT
Returns all rows from the left table.	Every row from the left table will be included.	Returns only the matching rows from the right table.	If no match is found, NULL is returned for right table columns.	Based on a related column using the ON condition.	Combines rows from two tables based on condition.	Use LEFT JOIN clause in your SELECT query.	Result includes all left rows with match or NULLs.

HOW LEFT JOIN WORKS



LEFT JOIN Result
(All rows from A + Matching rows from B)

- Returns all rows from the left table (A).
- Returns matching rows from the right table (B).
- If no match, NULL values are returned for B columns.
- Condition is specified using ON keyword.

EXAMPLE TABLES

Employees (Left Table)

EmpID (PK)	EmpName	DeptID	Salary
1	Alice	10	60000
2	Bob	20	55000
3	Charlie	10	70000
4	David	30	50000
5	Eva	40	65000

Departments (Right Table)

DeptID (PK)	DeptName	Location
10	IT	New York
20	HR	London
30	Finance	Delhi

Relation: Employees.DeptID → Departments.DeptID

Note: DeptID 40 in Employees does not exist in Departments, so NULL will be shown for Department columns.

SYNTAX

```
SELECT   column_list
FROM     LeftTable A
LEFT JOIN RightTable B
ON       A.common_column = B.common_column;
```

LEFT JOIN returns all rows from the left table (A) and the matching rows from the right table (B). If no match is found, NULL is returned for columns from the right table.

LEFT JOIN EXAMPLE QUERY

```
SELECT   e.EmpID, e.EmpName, e.DeptID, e.Salary,
         d.DeptName, d.Location
FROM     Employees e
LEFT JOIN Departments d
ON       e.DeptID = d.DeptID;
```

RESULT (LEFT JOIN OUTPUT)

EmpID	EmpName	DeptID	Salary	DeptName	Location
1	Alice	10	60000	IT	New York
2	Bob	20	55000	HR	London
3	Charlie	10	70000	IT	New York
4	David	30	50000	Finance	Delhi
5	Eva	40	65000	NULL	NULL

All 5 rows from Employees (left table) are returned.
DeptID 40 has no match, so DeptName and Location are NULL.

VISUAL EXPLANATION

Employees (Left Table)

EmpID	EmpName	DeptID
1	Alice	10
2	Bob	20
3	Charlie	10
4	David	30
5	Eva	40

Departments (Right Table)

DeptID	DeptName
10	IT
20	HR
30	Finance

Result = All rows from Left + Matching from Right
(No match → NULL for Right Table columns)

WHEN TO USE LEFT JOIN?

- When you want all data from the left table.
- When related data from the right table is optional.
- When some left rows may not have a match.
- When building reports with missing related data.

NULL VALUES IN RESULT

If no match is found in right table, its columns will contain NULL.

Example

EmpID	EmpName	DeptID	DeptName	Location
5	Eva	40	NULL	NULL

TYPES OF JOIN (QUICK VIEW)

INNER JOIN	Returns only matching rows from both tables.
LEFT JOIN	Returns all rows from left table and matching from right.
RIGHT JOIN	Returns all rows from right table and matching from left.
FULL OUTER JOIN	Returns all rows from both tables (match + unmatched).

PERFORMANCE TIPS

- Create indexes on join columns (EmpID, DeptID).
- Avoid SELECT * ; select only required columns.
- Use appropriate data types for join columns.
- Use WHERE clause after LEFT JOIN carefully (else it may act like INNER JOIN).

BEST PRACTICES

- Always use table aliases (e, d) for readability.
- Clearly specify ON condition.
- Check for NULLs in right table columns when processing results.
- Test with sample data including unmatched rows.

COMMON MISTAKES

- Forgetting ON condition (causes Cartesian product).
- Using WHERE on right table columns (turns into INNER JOIN).
- Not handling NULL values in application logic.
- Selecting unnecessary columns (slows performance).

QUICK SUMMARY

- LEFT JOIN returns all rows from the left table.
- It returns matching rows from the right table.
- If no match → NULL values in right table columns.
- Useful when related data is optional.
- Helps in building complete and accurate reports.

SQL RIGHT JOIN

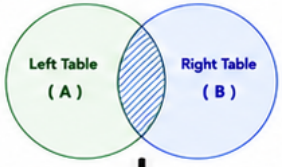
Get All Rows from Right Table and Matching Rows from Left Table

RIGHT JOIN returns all rows from the right table, and the matching rows from the left table.
If no match is found, NULL values are returned for columns from the left table.

KEY CONCEPTS

RIGHT JOIN	ALL RIGHT ROWS	MATCHING LEFT	NULL VALUES	RELATIONSHIP	JOIN	QUERY	RESULT
Returns all rows from right table.	Every row from the right table will be included.	Returns only the matching rows from the left table.	If no match is found, NULL is returned for left table columns.	Based on a related column using the ON condition.	Combines rows from two tables based on condition.	Use RIGHT JOIN clause in your SELECT query.	Result includes all right rows with match or NULLs.

HOW RIGHT JOIN WORKS



RIGHT JOIN Result
(All rows from B + Matching rows from A)

- Returns all rows from the right table (B).
- Returns matching rows from the left table (A).
- If no match, NULL values are returned for A columns.
- Condition is specified using ON keyword.

EXAMPLE TABLES

Employees (Left Table)

EmpID (PK)	EmpName	DeptID	Salary
1	Alice	10	60000
2	Bob	20	55000
3	Charlie	10	70000
4	David	30	50000

Departments (Right Table)

DeptID (PK)	DeptName	Location
10	IT	New York
20	HR	London
30	Finance	Delhi
40	Marketing	Sydney

Relation: Employees.DeptID → Departments.DeptID
Note: DeptID 40 in Departments does not exist in Employees, so NULL will be shown for Employees columns.

SYNTAX

```
SELECT  column_list
FROM    LeftTable A
RIGHT JOIN RightTable B
ON      A.common_columm = B.common_column;
```

RIGHT JOIN returns all rows from the right table (B) and the matching rows from the left table (A). If no match is found, NULL is returned for columns from the left table.

RIGHT JOIN EXAMPLE QUERY

```
SELECT  e.EmpID, e.EmpName, e.DeptID, e.Salary,
        d.DeptName, d.Location
FROM    Employees e
RIGHT JOIN Departments d
ON      e.DeptID = d.DeptID;
```

RESULT (RIGHT JOIN OUTPUT)

EmpID	EmpName	DeptID	Salary	DeptName	Location
1	Alice	10	60000	IT	New York
2	Bob	20	55000	HR	London
4	David	30	50000	Finance	Delhi
NULL	NULL	NULL	NULL	Marketing	Sydney

All 4 rows from Departments (right table) are returned.
DeptID 40 has no match, so EmpID, EmpName, Salary are NULL.

VISUAL EXPLANATION

Employees (Left Table)

EmpID	EmpName	DeptID	Salary
1	Alice	10	60000
2	Bob	20	55000
3	Charlie	10	70000
4	David	30	50000

Departments (Right Table)

DeptID	DeptName	Location
10	IT	New York
20	HR	London
30	Finance	Delhi
40	Marketing	Sydney

Result = All rows from Right + Matching from Left
(No match → NULL for Left Table columns)

WHEN TO USE RIGHT JOIN?

- When you want all data from the right table.
- When related data from the left table is optional.
- When some right rows may not have a match.
- When building reports with master data in right table.

NULL VALUES IN RESULT

If no match is found in left table, its columns will contain NULL.

Example (DeptID 40)

EmpID	EmpName	DeptID	Salary	DeptName	Location
NULL	NULL	NULL	NULL	Marketing	Sydney

TYPES OF JOIN (QUICK VIEW)

INNER JOIN	Returns only matching rows from both tables.
LEFT JOIN	Returns all rows from left table and matching from right.
RIGHT JOIN	Returns all rows from right table and matching from left.
FULL OUTER JOIN	Returns all rows from both tables (matched + unmatched).

QUERY TIPS

- Use RIGHT JOIN when right table is the main table.
- Use table aliases for readability.
- Make sure ON condition is correct for proper results.
- Check for NULLs in left table columns to handle unmatched rows.

BEST PRACTICES

- Index join columns (e.g., DeptID) for better performance.
- Use specific column names instead of *.
- Filter data after JOIN to reduce result set.
- Test queries with sample data including no-match cases.

COMMON MISTAKES

- Forgetting ON condition (causes cartesian product).
- Using WHERE instead of ON (turns into INNER JOIN).
- Not handling NULL values in left table columns.
- Selecting unnecessary columns (affects performance).

QUICK SUMMARY

- RIGHT JOIN returns all rows from right table.
- It returns matching rows from left table.
- If no match → NULL values in left table columns.
- Useful for optional relationships and master data.
- Helps in building complete and accurate reports.

SQL FULL JOIN

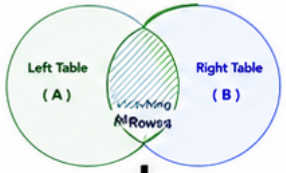
Get All Records from Both Tables – Matched and Unmatched!

FULL JOIN (or FULL OUTER JOIN) returns all rows when there is a match in either left or right table.
If there is no match, NULL values are returned for the missing side.

KEY CONCEPTS

FULL JOIN	ALL RECORDS	LEFT + RIGHT	MATCHING ROWS	NULL VALUES	MERGE	QUERY	RESULTS
Returns all rows from both tables.	Includes matched and unmatched rows from both sides.	Combines all rows from left table and right table.	Returns matching rows based on the join condition.	Unmatched rows contain NULL for the missing side.	Merges data from both tables into one result set.	Use FULL JOIN clause in your SELECT query.	Result includes all rows from both tables.

HOW FULL JOIN WORKS



FULL JOIN Result
All rows from A + All rows from B

- Matched rows appear once.
- Unmatched rows from left table → NULL in right columns.
- Unmatched rows from right table → NULL in left columns.

EXAMPLE TABLES

Employees (Left Table)

EmpID (PK)	EmpName	DeptID	Salary
1	Alice	10	60000
2	Bob	20	55000
3	Charlie	10	70000
4	David	30	50000

Departments (Right Table)

DeptID (PK)	DeptName	Location
10	IT	New York
20	HR	London
30	Finance	Delhi
40	Marketing	Sydney

Relation: Employees.DeptID → Departments.DeptID
Note: DeptID 40 exists only in Departments.
DeptID 30 has no matching employee.

SYNTAX

```
SELECT  column_list
FROM    LeftTable A
FULL JOIN RightTable B
ON      A.common_column = B.common_column;
```

FULL JOIN returns all rows from both tables.
Matched rows have data in both sides.
Unmatched rows have NULLs for the missing side.

EXAMPLE QUERY

```
SELECT  e.EmpID, e.EmpName, e.DeptID, e.Salary,
        d.DeptName, d.Location
FROM    Employees e
FULL JOIN Departments d
ON      e.DeptID = d.DeptID;
```

RESULT (FULL JOIN OUTPUT)

EmpID	EmpName	DeptID	Salary	DeptName	Location
1	Alice	10	60000	IT	New York
2	Bob	20	55000	HR	London
3	Charlie	10	70000	IT	New York
4	David	30	50000	Finance	Delhi
NULL	NULL	40	NULL	Marketing	Sydney

All records from both tables are returned.
Unmatched rows contain NULLs in the missing side.

VISUAL EXPLANATION

Employees (Left Table)

EmpID	EmpName	DeptID	Salary
1	Alice	10	60000
2	Bob	20	55000
3	Charlie	10	70000
4	David	30	50000

Departments (Right Table)

DeptID	DeptName	Location
10	IT	New York
20	HR	London
30	Finance	Delhi
40	Marketing	Sydney

Result = All Rows from Left + All Rows from Right
(Matched + Unmatched, with NULLs for missing side)

WHEN TO USE FULL JOIN?

- When you need all data from both tables.
- When you want to see unmatched rows from both sides.
- For data comparison and data auditing.
- For reports where no data should be missed.

NULL VALUE EXAMPLES

Unmatched rows will have NULL in columns from the other table.

Example

EmpID	EmpName	DeptID	DeptName
4	David	30	Finance
NULL	NULL	40	Marketing

TYPES OF JOIN (QUICK VIEW)

INNER JOIN	Returns only matching rows from both tables.
LEFT JOIN	Returns all rows from left and matching from right.
RIGHT JOIN	Returns all rows from right and matching from left.
FULL JOIN	Returns all rows from both (matched + unmatched).

QUERY TIPS

- Use FULL JOIN when you need complete data from both tables.
- Always specify proper ON condition.
- Check for NULLs to find unmatched rows.
- Use table aliases for better readability.

BEST PRACTICES

- Index join columns (e.g., DeptID) for better performance.
- Avoid SELECT *; select only required columns.
- Test with sample data including unmatched rows.
- Use WHERE clause carefully (may turn FULL JOIN into INNER JOIN if not handled properly).

COMMON MISTAKES

- Forgetting ON condition (causes Cartesian product).
- Adding WHERE filter on right table columns (turns into LEFT JOIN).
- Assuming NULL means zero or empty.
- Not handling unmatched rows in reports properly.

QUICK SUMMARY

- FULL JOIN returns all records from both tables.
- Includes matched and unmatched rows.
- Unmatched rows contain NULL in the missing side.
- Great for data comparison, auditing and reporting.
- Helps to get complete picture of data.

SQL UNION

Combine Results from Multiple SELECT Queries

The UNION operator is used to combine the result-set of two or more SELECT statements into a single result.

COMBINE TABLES	REMOVE DUPLICATES	MERGE RESULTS	SELECT	QUTRY	OUTPUT
Combine results from multiple SELECT queries.	UNION removes duplicate rows.	Merges rows into a single result set.	Works with SELECT statements only.	Write separate queries and combine them.	Returns the final combined result.

1. UNION (Removes Duplicates)

UNION combines rows from both queries and removes duplicate rows automatically.

Employees Table

EmpID	Name	City
1	Alice	Delhi
2	Bob	Mumbai
3	Charlie	Pune

+

Customers Table

EmpID	Name	City
3	Charlie	Pune
4	David	Delhi
5	Eve	Kolkata

SYNTAX

```
SELECT column1, column2, ...
FROM table1
UNION
SELECT column1, column2, ...
FROM table2;
```

Both SELECT queries must have the same number of columns and compatible data types.

UNION RESULT (Duplicates Removed)

EmpID	Name	City
1	Alice	Delhi
2	Bob	Mumbai
3	Charlie	Pune
4	David	Delhi
5	Eve	Kolkata

Duplicate row (Charlie, Pune) appears only once.

2. UNION ALL (Keeps Duplicates)

UNION ALL combines all rows from both queries including duplicates.

Employees Table

EmpID	Name	City
1	Alice	Delhi
2	Bob	Mumbai
3	Charlie	Pune

+

Customers Table

EmpID	Name	City
3	Charlie	Pune
4	David	Delhi
5	Eve	Kolkata

SYNTAX

```
SELECT column1, column2, ...
FROM table1
UNION ALL
SELECT column1, column2, ...
FROM table2;
```

UNION ALL is faster because it does not remove duplicates.

UNION ALL RESULT (Duplicates Kept)

EmpID	Name	City
1	Alice	Delhi
2	Bob	Mumbai
3	Charlie	Pune
3	Charlie	Pune
4	David	Delhi
5	Eve	Kolkata

Duplicate row (Charlie, Pune) appears again.

UNION vs UNION ALL

Feature	UNION	UNION ALL
Duplicates	Removes duplicates	Keeps all duplicates
Performance	Slower (extra work to remove duplicates)	Faster (no duplicate check)
Use Case	When you want unique results	When you need all records
Default	Sorts & removes duplicates	Does not sort & does not remove

USE CASES

- **UNION:** Combine similar data from multiple sources and get unique values (e.g., list of all cities).
- **UNION ALL:** Combine log data, transactions, or reports where duplicates are important.
- Merge data from different tables with same structure into a single output.

IMPORTANT NOTES

- The number of columns in all SELECT statements must be same.
- The data types of corresponding columns must be compatible.
- Column names in the output are taken from the first SELECT.
- ORDER BY can be used only at the end of the final query.

EXAMPLE QUERY

```
-- Using UNION
SELECT Name, City FROM Employees
UNION
SELECT Name, City FROM Customers;

-- Using UNION ALL
SELECT Name, City FROM Employees
UNION ALL
SELECT Name, City FROM Customers;
```

KEY POINTS

- UNION = Unique Results
- UNION ALL = All Results
- Both require same columns
- Great for reporting & data consolidation
- Choose based on your requirement

COMMON MISTAKES

- Using different number of columns in SELECT statements.
- Using incompatible data types for same column. column position.
- Forgetting that UNION removes plicates (may hide data).
- Using ORDER BY inside individual SELECT((not allowed).

BEST PRACTICES

- Use UNION ALL when duplicates are acceptable.
- Use UNION when you need distinct results.
- Add ORDER BY at the end of the final query.
- Ensure proper indexing for better performance.

QUICK SUMMARY

Keyword	What it Does
UNION	Combines results and removes duplicates.
UNION ALL	Combines results and keeps all duplicates.

Use UNION for unique data.
Use UNION ALL for complete data with better performance.

Remember: UNION = Combine + Remove Duplicates → Clean, Unique Output | UNION ALL = Combine + Keep Duplicates → Complete Output, Faster Performance

SQL SUBQUERIES

Get Data Using the Result of Another Query

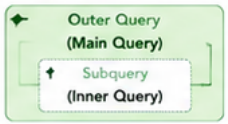
A subquery (or nested query) is a query inside another query.
The inner query runs first, and its result is used by the outer query.

KEY CONCEPTS

SUBQUERY	NESTED QUERY	SELECT	WHERE	FROM	IN	EXISTS	RESULT
A query inside another query.	The inner query is nested in the outer query.	Retrieves data from tables.	Filters records based on conditions.	Specifies the table(s) to retrieve data from.	Checks if a value matches any value in a list (subquery).	Checks if the subquery returns at least one row.	Returns the final result to the outer query.

WHAT IS A SUBQUERY?

A subquery is a SELECT statement inside another SQL statement.
It can be used with SELECT, INSERT, UPDATE, DELETE.



EXAMPLE TABLES

Employees

EmpID	EmpName	DeptID	Salary
1	Alice	10	60000
2	Bob	20	55000
3	Charlie	10	70000
4	David	30	50000
5	Eva	20	45000

Departments

DeptID	DeptName
10	IT
20	HR
30	Finance

TYPES OF SUBQUERIES WITH EXAMPLES

1. SUBQUERY WITH IN

Use **IN** to check if a value matches any value returned by the subquery.

```
SELECT EmpName, Salary
FROM  Employees
WHERE DeptID IN (SELECT DeptID
                 FROM Departments
                 WHERE DeptName = 'HR');
```

Result

EmpName	Salary
Bob	55000
Eva	45000

✔ Returns employees who belong to the **HR department**.

2. SUBQUERY WITH EXISTS

Use **EXISTS** to check if the subquery returns at least one row.

```
SELECT EmpName, Salary
FROM  Employees e
WHERE EXISTS (SELECT 1
              FROM Departments d
              WHERE d.DeptID = e.DeptID
               AND d.DeptName = 'IT');
```

Result

EmpName	Salary
Alice	60000

✔ Returns employees who belong to the IT department.

3. SCALAR SUBQUERY (Single Value)

Returns a single value to be used in a condition.

```
SELECT EmpName, Salary
FROM  Employees
WHERE Salary > (SELECT AVG(Salary)
                FROM Employees);
```

Result

EmpName	Salary
Charlie	70000
Alice	60000

✔ Returns employees with salary greater than the average salary.

4. SUBQUERY IN FROM (Derived Table)

Use a subquery in the **FROM clause.**

```
SELECT DeptName, AvgSalary
FROM (SELECT DeptID, AVG(Salary) AS AvgSalary
      FROM Employees
      GROUP BY DeptID) AS DeptAvg
JOIN Departments d ON d.DeptID = DeptAvg.DeptID;
```

Result

EmptName	AygSalary
IT	65000
HR	50000
Finance	50000

✔ Shows average salary of employees in each department.

SUBQUERY SYNTAX

General Syntax

```
SELECT column(s)
FROM  table_name
WHERE column_name operator (SELECT column(s)
                            FROM  table_name
                            WHERE condition);
```

Existence Check Syntax

```
SELECT column(s)
FROM  table_name
WHERE EXISTS  (SELECT 1
               FROM table_name
               WHERE condition);
```

COMMON OPERATORS WITH SUBQUERIES

Operator	Description
IN	Matches any value in the subquery result.
NOT IN	Does not match any value in the subquery result.
EXISTS	True if subquery returns one or more rows.
NOT EXISTS	True if subquery returns no rows.
=, >, <, >=, <=	Used with scalar subqueries (single value).

SUBQUERY VS JOIN

Subquery	Join
Easier to write for complex filters.	Better performance on large datasets.
Good for small to medium data.	Best for combining table data.
Can be nested multiple levels.	Usually used for two or more tables.

NESTED SUBQUERY (MULTI-LEVEL)

A subquery inside another subquery.

```
SELECT EmpName
FROM  Employees
WHERE DeptID IN (SELECT DeptID
                 FROM Departments
                 WHERE DeptID IN (SELECT DeptID
                                  FROM Employees
                                  WHERE Salary > 60000));
```

PRACTICAL USE CASES

- Filter data using complex conditions.
- Find records based on aggregated values.
- Check existence of related data.
- Use derived tables for advanced reporting.

COMMON MISTAKES

- Forgetting to use parentheses ().
- Using multiple columns in IN with mismatched data types.
- Using = with subquery that returns multiple rows.
- Not considering performance for large datasets.

QUICK SUMMARY

Keyword	Purpose
Subquery	Inner query inside another query.
IN	Check value in a list returned by subquery.
EXISTS	Check if subquery returns any rows.
Scalar Subquery	Returns a single value.
Nested Query	Subquery inside another subquery.
Result	Final output returned by outer query.

PRO TIP: Use subqueries for readability and complex conditions. Use JOINs for better performance on large datasets.

Remember: Subquery runs first, Outer query runs next, Final result is returned.

EXISTS OPERATOR

Filter Rows Based on Existence of Related Data

The EXISTS operator is used with a subquery to check if the subquery returns at least one row. It returns **TRUE** or **FALSE** (Boolean) and is often used in **WHERE** or **HAVING** clause.

KEY CONCEPTS

- **EXISTS** — Returns TRUE if the subquery returns one or more rows.
- **NOT EXISTS** — Returns TRUE if the subquery returns no rows.
- **SUBQUERY** — An inner query inside another query.
- **FILTER** — Filters rows based on the existence of related data.
- **RELATED DATA** — Checks relationship between tables without JOIN.
- **BOOLEAN** — EXISTS returns TRUE or FALSE (Boolean result).
- **RESULT / ROWS** — Only rows satisfying the condition are returned.

HOW EXISTS WORKS

Outer Query (Main Query) → EXISTS (Subquery) → Returns TRUE / FALSE

If subquery returns at least one row
⇒ **TRUE** (Row is included)

If subquery returns no rows
⇒ **FALSE** (Row is excluded)

SAMPLE TABLES

Employees

EmpID	EmpName	DeptID	Salary
1	Alice	10	60000
2	Bob	20	55000
3	Charlie	10	70000
4	David	30	50000
5	Eva	40	45000

Departments

DeptID	DeptName	Location
10	IT	New York
20	HR	London
30	Finance	Delhi

EXISTS is usually faster than IN for subqueries on large datasets because it stops searching after finding the first match.

1. EXISTS (Returns TRUE if subquery returns any row)

Get employees who belong to a department located in 'New York'.

```
SELECT EmpName, DeptID
FROM Employees e
WHERE EXISTS (
      SELECT 1
      FROM Departments d
      WHERE d.DeptID = e.DeptID
        AND d.Location = 'New York'
);
```

How it works:

- For each employee, the subquery checks if a matching department exists in New York.
- If found, EXISTS returns TRUE and the employee is included.

Result

EmpName	DeptID	Salary
Alice	10	60000

Only Alice belongs to a department located in New York.

2. NOT EXISTS (Returns TRUE if subquery returns no rows)

Get employees who do NOT belong to any department located in 'New York'.

```
SELECT EmpName, DeptID
FROM Employees e
WHERE NOT EXISTS (
      SELECT 1
      FROM Departments d
      WHERE d.DeptID = e.DeptID
        AND d.Location = 'New York'
);
```

How it works:

- For each employee, the subquery checks for department in New York.
- If NO row is found, NOT EXISTS returns TRUE and the employee is included.

Result

EmpName	DeptID	Salary
Bob	20	55000
Charlie	10	70000
David	30	50000
Eva	40	45000

All employees except Alice are returned.

EXAMPLE QUERIES

3. EXISTS with Correlated Subquery

Get departments that have at least one employee.

```
SELECT DeptID, DeptName
FROM Departments d
WHERE EXISTS (
     SELECT 1
     FROM Employees e
     WHERE e.DeptID = d.DeptID
);
```

Result

DeptID	DeptName
10	IT
20	HR
30	Finance

4. NOT EXISTS with Correlated Subquery

Get departments that have no employees.

```
SELECT DeptID, DeptName
FROM Departments d
WHERE NOT EXISTS (
     SELECT 1
     FROM Employees e
     WHERE e.DeptID = d.DeptID
);
```

Result

DeptID	DeptName

No rows returned

5. EXISTS with Additional Condition

Employees who are in departments with salary budget > 55000.

```
SELECT EmpName, Salary
FROM Employees e
WHERE EXISTS (
     SELECT 1
     FROM Departments d
     WHERE d.DeptID = e.DeptID
       AND d.Budget > 55000
);
```

Result

EmpName	Salary
Alice	60000
Charlie	70000

6. NOT EXISTS with Additional Condition

Employees who are NOT in departments with salary budget > 55000.

```
SELECT EmpName, Salary
FROM Employees e
WHERE NOT EXISTS (
    SELECT 1
    FROM Departments d
    WHERE d.DeptID = e.DeptID
      AND d.Budget > 55000
);
```

Result

EmpName	Salary
Bob	55000
David	50000
Eva	45000

EXISTS VS IN

Feature	EXISTS	IN
Returns	TRUE/FALSE	Value list match
Stops Early	Yes (after first match)	No
Handles NULL	Better	May return unexpected results
Performance	Faster for large subqueries	Slower for large subqueries
Use Case	Check existence of related rows	Check value membership

EXISTS VS NOT EXISTS EXAMPLE

EXISTS	Find customers who have at least one order.
NOT EXISTS	Find customers who have no orders.

COMMON USE CASES

- Check if related records exist in another table.
- Validate parent-child relationships.
- Find missing related records.
- Apply complex filters with correlated subqueries.

BEST PRACTICES

- Use EXISTS instead of IN for large subqueries.
- Use NOT EXISTS to find missing related data.
- Keep subqueries correlated for accuracy.
- Ensure proper indexing for better performance.

SYNTAX QUICK REFERENCE

EXISTS	NOT EXISTS
WHERE EXISTS (subquery)	WHERE NOT EXISTS (subquery)

IMPORTANT NOTES

- EXISTS returns TRUE if the subquery returns at least one row.
- NOT EXISTS returns TRUE if the subquery returns no rows.
- The SELECT list inside EXISTS can be any value (commonly SELECT 1).

COMMON MISTAKES

- Forgetting to correlate subquery with outer query.
- Using IN when subquery returns NULL values.
- Not considering performance on large datasets.

QUICK SUMMARY

EXISTS → TRUE if subquery returns at least one row.

NOT EXISTS → TRUE if subquery returns no rows.

SQL ANY & ALL

Compare a Value with Subquery Results

ANY and ALL operators are used with a subquery to compare a value with one or more values returned by the subquery.

KEY CONCEPTS

- **ANY** — Returns TRUE if the condition is true for ANY one value.
- **ALL** — Returns TRUE if the condition is true for ALL values.
- **COMPARISON** — Works with comparison operators: >, <, >=, <=, =, <>, !=.
- **SUBQUERY** — Uses the result of an inner query for comparison.
- **GREATER THAN** — >, >= : Checks if value is greater than ANY/ALL subquery values.
- **LESS THAN** — <, <= : Checks if value is less than ANY/ALL subquery values.
- **CONDITIONS** — Combine with WHERE clause to filter records.
- **RESULTS** — Returns rows that satisfy the comparison condition.

WHAT ARE ANY & ALL?

ANY checks if the condition is true for ANY one value in the subquery result.

ALL checks if the condition is true for ALL values in the subquery result.

ANY (SOME)	ALL
One or more match → TRUE	All must match → TRUE

SAMPLE TABLES

Employees

EmpID	EmpName	DeptID	Salary
1	Alice	10	60000
2	Bob	20	55000
3	Charlie	10	70000
4	David	30	50000
5	Eva	20	45000

Departments

DeptID	DeptName	Budget
10	IT	65000
20	HR	50000
30	Finance	40000

ANY = At least one value satisfies the condition.

ALL = All values must satisfy the condition.

1. ANY OPERATOR (TRUE if ANY one value satisfies the condition)

Example 1: Salary > ANY (Subquery)

Find employees whose salary is greater than ANY department budget.

```
SELECT EmpName, Salary
FROM Employees
WHERE Salary > ANY
      (SELECT Budget FROM Departments);
```

Subquery Result (Budgets)

Budget
65000
50000
40000

Condition: Salary > ANY (65000, 50000, 40000) — TRUE if Salary > at least one budget

Result

EmpName	Salary
Alice	60000
Charlie	70000

Employees with salary > at least one budget (50000 or 40000) are returned.

Example 2: Salary < ANY (Subquery)

Find employees whose salary is less than ANY department budget.

```
SELECT EmpName, Salary
FROM Employees
WHERE Salary < ANY
      (SELECT Budget FROM Departments);
```

Condition: Salary < ANY (65000, 50000, 40000) — TRUE if Salary < at least one budget

Result

EmpName	Salary
Bob	55000
David	50000
Eva	45000

Employees with salary less than at least one budget are returned.

2. ALL OPERATOR (TRUE only if condition is true for ALL values)

Example 1: Salary > ALL (Subquery)

Find employees whose salary is greater than ALL department budgets.

```
SELECT EmpName, Salary
FROM Employees
WHERE Salary > ALL
      (SELECT Budget FROM Departments);
```

Subquery Result (Budgets)

Budget
65000
50000
40000

Condition: Salary > ALL (65000, 50000, 40000) — TRUE if Salary > every budget

Result

EmpName	Salary
Charlie	70000

Only employees with salary greater than all budgets (i.e., > 65000) are returned.

Example 2: Salary < ALL (Subquery)

Find employees whose salary is less than ALL department budgets.

```
SELECT EmpName, Salary
FROM Employees
WHERE Salary < ALL
      (SELECT Budget FROM Departments);
```

Condition: Salary < ALL (65000, 50000, 40000) — TRUE if Salary < every budget

Result

EmpName	Salary
No rows returned	

No employeee has salary less than all budgets (less than 40000).

COMPARISON OPERATORS WITH ANY & ALL

Operator	Meaning with ANY	Meaning with ALL
>	Greater than ANY value	Greater than ALL values
>=	Greater than or equal to ANY value	Greater than or equal to ALL values
<	Less than ANY value	Less than ALL values
<=	Less than or equal to ANY value	Less than or equal to ALL values
=	Equal to ANY value	Equal to ALL values
<>	Not equal to ANY value	Not equal to ALL values

ANY vs ALL (QUICK COMPARISON)

Feature	ANY	ALL
Logic	OR logic (One match is enough)	AND logic (All must match)
Condition Check	True if condition is true for any one value	True if condition is true for every value
Result Set	Larger	Usually smaller
Use Case	When any match is sufficient	When all must satisfy the condition

MORE PRACTICAL EXAMPLES

1. > ANY Example

Employees with salary greater than ANY salary in DeptID = 20

```
SELECT EmpName, Salary
FROM Employees
WHERE Salary > ANY
      (SELECT Salary FROM Employees
       WHERE DeptID = 20);
```

DeptID = 20 Salaries

Salary
55000
45000

Result

EmpName	Salary
Alice	60000
Charlie	70000

2. > ALL Example

Employees with salary greater than ALL salary in DeptID = 20

```
SELECT EmpName, Salary
FROM Employees
WHERE Salary > ALL
      (SELECT Salary FROM Employees
       WHERE DeptID = 20);
```

DeptID = 20 Salaries

Salary
55000
45000

Result

EmpName	Salary
Alice	60000
Charlie	70000

3. < ANY Example

Employees with salary less than ANY salary in DeptID = 10

```
SELECT EmpName, Salary
FROM Employees
WHERE Salary < ANY
      (SELECT Salary FROM Employees
       WHERE DeptID = 10);
```

DeptID = 10 Salaries

Salary
60000
70000

Result

EmpName	Salary
Bob	55000
David	50000
Eva	45000

4. < ALL Example

Employees with salary less than ALL salary in DeptID = 10

```
SELECT EmpName, Salary
FROM Employees
WHERE Salary < ALL
      (SELECT Salary FROM Employees
       WHERE DeptID = 10);
```

DeptID = 10 Salaries

Salary
60000
70000

Result

EmpName	Salary
Eva	45000

COMMON USE CASES

- Compare values with a list returned by subquery.
- Filter records based on dynamic conditions.
- Use in WHERE clause for advanced filtering.
- Useful in reports, analytics and validations.

SYNTAX QUICK REFERENCE

ANY

```
SELECT column(s) FROM table
WHERE column operator ANY (subquery);
```

ALL

```
SELECT column(s) FROM table
WHERE column operator ALL (subquery);
```

IMPORTANT NOTES

- ANY returns TRUE if the condition is true for at least one value.
- ALL returns TRUE only if the condition is true for all values.
- If subquery returns no rows:
  ANY → FALSE | ALL → TRUE

COMMON MISTAKES

- Using ANY when you need all conditions to match.
- Using ALL when only one match is required.
- Not considering results when subquery returns no rows.
- Forgetting NULL values in subquery (they can affect results).

QUICK SUMMARY

Keyword	Meaning	Example (Concept)
ANY	True if condition is true for any one value	Salary > ANY (subquery)
ALL	True if condition is true for all values	Salary > ALL (subquery)

PRO TIP: Use ANY for flexible matching and ALL for strict matching. Always test subqueries separately first!

Remember: ANY = One match is enough, ALL = All must match.

Common Table Expressions (CTE)

Write Better, Cleaner and More Readable SQL Queries

CTE (Common Table Expression) is a temporary result set that you define with the WITH clause and use in a SELECT, INSERT, UPDATE or DELETE statement.

KEY CONCEPTS

WITH	CTE	REUSABLE	RECURSIVE	READABLE QUERY	TEMPORARY RESULT	OPTIMIZATION
Starts the CTE definition.	Temporary named result set.	Use the CTE multiple times in the same query.	CTE can reference itself for hierarchical data.	Makes complex queries easier to read and maintain.	Exists only for the execution of the query.	Helps in better query planning & performance.

WHY USE CTE?

- Improves query readability
- Breaks complex queries into smaller parts
- Makes query easier to maintain
- Reusable within the same query
- Great for recursive/hierarchical data
- Can improve optimization in many scenarios

SAMPLE TABLE

Employees

EmpID	EmpName	ManagerID	Salary
1	Alice	NULL	70000
2	Bob	1	60000
3	Charlie	1	55000
4	David	2	50000
5	Eva	2	48000

1. BASIC CTE EXAMPLE

Get employees with salary greater than 55000 using CTE.

```
WITH HighSalary AS (
  SELECT EmpID, EmpName, Salary
  FROM Employees
  WHERE Salary > 55000
)
SELECT * FROM HighSalary;
```

Result

EmpID	EmpName	Salary
1	Alice	70000
2	Bob	60000

- CTE acts like a temporary table (HighSalary) for this query.

2. MULTIPLE CTEs (Reusable)

Use CTE multiple times in the same query.

```
WITH DeptEmp AS (
  SELECT ManagerID, COUNT(*) AS TotalEmp
  FROM Employees
  WHERE ManagerID IS NOT NULL
  GROUP BY ManagerID
)
SELECT e.EmpID, e.EmpName, d.TotalEmp
FROM Employees e
LEFT JOIN DeptEmp d ON e.EmpID = d.ManagerID;
```

Result

EmpID	EmpName	TotalEmp
1	Alice	2
2	Bob	2
3	Charlie	NULL
4	David	NULL
5	Eva	NULL

- CTE (DeptEmp) is defined once and used in the main query.

3. RECURSIVE CTE (Hierarchical Data)

Get the organization hierarchy (Manager → Employees).

```
WITH RECURSIVE EmpHierarchy AS (
  SELECT EmpID, EmpName, ManagerID, 0 AS Level
  FROM Employees
  WHERE ManagerID IS NULL
  UNION ALL
  SELECT e.EmpID, e.EmpName, e.ManagerID,
         h.Level + 1
  FROM Employees e
  JOIN EmpHierarchy h ON e.ManagerID = h.EmpID
)
SELECT EmpID, EmpName, ManagerID, Level
FROM EmpHierarchy;
```

Result

EmpID	EmpName	ManagerID	Level
1	Alice	NULL	0
2	Bob	1	1
3	Charlie	1	1
4	David	2	2
5	Eva	2	2

- Recursive CTE calls itself to walk through the hierarchy.

CTE vs Subquery

Feature	CTE	Subquery
Readability	Better (clean & clear)	Can be hard to read
Reusability	Yes (use multiple times)	No (write again)
Recursive	Supported	Not supported
Complex Query	Easier to manage	Becomes messy
Performance	Often better	Can be slower

COMMON USE CASES

- Reporting and data analysis
- Complex joins and aggregations
- Hierarchical / tree structures (org chart, categories)
- Data transformation steps
- Business logic with multiple stages

SYNTAX TEMPLATE

```
WITH CTE_Name (column1, column2, ...) AS (
   SELECT column1, column2, ...
   FROM table_name
   WHERE condition
)
SELECT column1, column2, ...
FROM CTE_Name
WHERE condition;
```

PERFORMANCE & OPTIMIZATION TIPS

- Use CTE for better readability and maintainability.
- Prefer CTE over deeply nested subqueries.
- Add proper indexes on columns used in joins/filters.
- For large datasets, test execution plan.
- Avoid unnecessary columns in CTE to reduce processing.

KEY ADVANTAGES

- Cleaner and readable SQL code
- Breaks query into logical blocks
- Reusable within the same query
- Supports recursion
- Helps manage complex queries

COMMON MISTAKES

- Forgetting to use WITH keyword
- Not giving proper column names
- Using CTE where simple query is enough
- Expecting CTE to persist (it's temporary)
- Ignoring indexes for large data

QUICK SUMMARY

Full Name	Common Table Expression
Keyword	WITH
Purpose	Temporary named result set
Supports	Normal & Recursive CTE
Main Benefit	Readable, reusable, maintainable queries

FINAL NOTES

CTE is a powerful feature in SQL that makes complex queries **simple, readable** and **reusable.**

Use it for better code quality and performance, especially in reporting and hierarchical data.

STORED PROCEDURES

Write Once, Reuse Many Times!

Stored Procedures are precompiled collections of SQL statements that can accept parameters, perform operations, and return results. They improve performance, security, and maintainability.

STORED PROCEDURE	CREATE PROCEDURE	CALL / EXECUTE	PARAMETERS	REUSABLE CODE	BUSINESS LOGIC	PERFORMANCE	AUTOMATION
A named block of SQL statements stored in the database.	Command to create a stored procedure in the database.	Command to execute a stored procedure with parameters.	Accept input values (IN, OUT, INOUT) for flexibility.	Write once and use many times. Easy to maintain.	Encapsulate complex business rules inside the database.	Precompiled & optimized execution for faster performance.	Ideal for scheduled tasks, jobs, and automated processes.

SAMPLE TABLE: Employees

EmpID	EmpName	DeptID	Salary
1	Alice	10	60000
2	Bob	20	55000
3	Charlie	10	70000
4	David	30	50000
5	Eva	20	45000

WHAT IS A STORED PROCEDURE?

- A saved SQL code that can be executed many times.
- Can accept parameters and return results.
- Helps in code reusability and modularity.
- Improves security by restricting direct table access.
- Stored in the database and executed on the server.

KEY BENEFITS

- Faster execution (precompiled execution plan).
- Reduced network traffic.
- Better security (grant execute, not direct access).
- Centralized business logic.
- Easy maintenance and updates.

PARAMETER TYPES

Type	Mode	Description	Example
IN	IN	Pass value to procedure	@EmpID INT
OUT	OUT	Return value from procedure	@Total INT OUTPUT
INOUT	INOUT	Pass value & return value	@Count INT INPUT OUTPUT

1. CREATE STORED PROCEDURE (WITH PARAMETER)

```
DELIMITER $$
CREATE PROCEDURE GetEmployeeByID (IN p_EmpID INT)
BEGIN
    SELECT EmpID, EmpName, DeptID, Salary
    FROM Employees
    WHERE EmpID = p_EmpID;
END $$
DELIMITER ;;
```

Explanation

- IN p_EmpID INT → Input parameter.
- Returns employee details for the given EmpID.

CALL / EXECUTE

```
CALL GetEmployeeByID(1);
```

Result

EmpID	EmpName	DeptID	Salary
1	Alice	10	60000

2. STORED PROCEDURE WITH MULTIPLE PARAMETERS

```
DELIMITER $$
CREATE PROCEDURE AddEmployee (
    IN p_EmpName VARCHAR(50),
    IN p_DeptID INT,
    IN p_Salary DECIMAL(10,2)
)
BEGIN
    INSERT INTO Employees (EmpName, DeptID, Salary)
    VALUES (p_EmpName, p_DeptID, p_Salary);
END $$
DELIMITER ;
```

Explanation

- Accepts multiple input parameters.
- Inserts a new employee record.

CALL / EXECUTE

```
CALL AddEmployee('Frank', 30, 48000);
```

✓ 1 row inserted.

3. STORED PROCEDURE WITH OUT PARAMETER

```
DELIMITER $$
CREATE PROCEDURE GetTotalEmployees (OUT p_Total INT)
BEGIN
    SELECT COUNT(*) INTO p_Total FROM Employees;
END $$
DELIMITER ;
```

CALL / EXECUTE

```
CALL GetTotalEmployees(@Total);
SELECT @Total AS TotalEmployees;
```

Result

TotalEmployees
5

OUT parameter stores the result in a variable that can be used after calling the procedure.

4. STORED PROCEDURE WITH INOUT PARAMETER

```
DELIMITER $$
CREATE PROCEDURE IncreaseSalary (INOUT p_Salary DECIMAL(10,2))
BEGIN
    SET p_Salary = p_Salary + 5000;
END $$
DELIMITER ;
```

Explanation

- INOUT → Value is passed in and modified.
- Updated value is returned.

```
SET @Sal = 60000;
CALL IncreaseSalary(@Sal);
SELECT @Sal AS NewSalary;
```

Result

NewSalary
65000

5. STORED PROCEDURE WITH RESULT SET

```
DELIMITER $$
CREATE PROCEDURE GetEmployeesByDept (IN p_DeptID INT)
BEGIN
    SELECT EmpID, EmpName, Salary FROM Employees
    WHERE DeptID = p_DeptID ORDER BY EmpName;
END $$
DELIMITER ;
```

CALL / EXECUTE

```
CALL GetEmployeesByDept(10);
```

Result (Multiple Rows)

EmpID	EmpName	Salary
1	Alice	60000
3	Charlie	70000

6. DROP STORED PROCEDURE

```
DROP PROCEDURE IF EXISTS GetEmployeeByID;
```

✓ Stored Procedure dropped successfully!

Note

- Use IF EXISTS to avoid errors.
- Keep procedures modular and well documented.

STORED PROCEDURE vs FUNCTIONS

Feature	Stored Procedure	Function
Returns	Result set / Values	Single value
Usage	CALL / EXECUTE	Used in SELECT
Transaction	Can handle transactions	Limited transactions
Parameters	IN, OUT, INOUT	IN only
Use Case	Complex operations	Calculations

BEST PRACTICES

- Use meaningful names.
- Keep procedures small and focused.
- Use parameters (avoid hardcoding).
- Handle errors using handlers.
- Comment your code.
- Test before using in production.

COMMON USE CASES

- Data validation before insert/update
- Complex report generation
- Bulk data operations
- Business rule enforcement
- Scheduled jobs and automation

COMMON MISTAKES

- Using hardcoded values.
- Making procedures too long and complex.
- Not handling errors properly.
- Returning unnecessary data.
- Not indexing related tables.

QUICK SUMMARY

Keyword	Purpose
CREATE PROCEDURE	Create a new stored procedure
CALL	Execute a stored procedure
IN / OUT / INOUT	Pass values to procedure
DROP PROCEDURE	Delete a stored procedure
DELIMITER	Define statement boundary

PRO TIP: Use stored procedures for complex operations and performance critical tasks.

SECURITY: Grant EXECUTE permission instead of direct table permissions.

PERFORMANCE: Stored Procedures are precompiled and optimized for speed.

SQL FUNCTIONS

Built-in Functions to Simplify Your Queries

SQL functions perform operations on data and return a single value. They make queries **shorter**, **readable** and **powerful**.

WHY USE FUNCTIONS?

- **Simplify Queries** — Do more with less code.
- **Improve Readability** — Easy to read and understand.
- **Save Time** — Built-in logic saves development time.
- **Consistent Results** — Return predictable and accurate output.
- **Reusable Logic** — Use functions again and again.
- **Better Performance** — Optimized for common tasks.

SAMPLE TABLE: Employees

EmpID	Name	City	Salary	JoinDate
1	alice	delhi	55000	2024-01-15
2	Bob	mumbai	62000	2024-03-10
3	Charlie	pune	47000	2024-02-20
4	David	delhi	58000	2024-04-05

WHAT IS A FUNCTION?

A function is a predefined block of code that accepts input (arguments), performs an operation and returns a value.

TYPES OF SQL FUNCTIONS

- String Functions
- Numeric Functions
- Date & Time Functions
- Aggregate Functions
- Conditional Functions
- Conversion Functions

SYNTAX

```
SELECT FunctionName(column)
FROM table_name;
```

COMMONLY USED SQL FUNCTIONS (MINIMAL OVERVIEW)

1. UPPER()

Converts text to uppercase.

```
SELECT UPPER(Name) AS UpperName
FROM Employees;
```

Name	UpperName
alice	ALICE
Bob	BOB
Charlie	CHARLIE
David	DAVID

2. LOWER()

Converts text to lowercase.

```
SELECT LOWER(Name) AS LowerName
FROM Employees;
```

Name	LowerName
alice	alice
Bob	bob
Charlie	charlie
David	david

3. LENGTH()

Returns the number of characters in a string.

```
SELECT LENGTH(Name) AS NameLength
FROM Employees;
```

Name	NameLength
alice	5
Bob	3
Charlie	7
David	5

4. ROUND()

Rounds a number to specified decimal places.

```
SELECT ROUND(Salary, 2) AS RoundedSalary
FROM Employees;
```

Salary	RoundedSalary
55000	55000.00
62000	62000.00
47000	47000.00
58000	58000.00

5. NOW()

Returns the current date and time.

```
SELECT NOW() AS CurrentDateTime;
```

Result

2025-05-25 14:30:45

Returns current date & time based on your system/server time zone.

6. CONCAT()

Joins two or more strings into one.

```
SELECT CONCAT(Name, ' - ', City) AS
NameCity FROM Employees;
```

Name	NameCity
alice	alice - delhi
Bob	Bob - mumbai
Charlie	Charlie - pune
David	David - delhi

Use CONCAT_WS(separator, str1, str2, ...) to add separator between strings.

7. IFNULL()

Returns a value if the expression is NULL.

```
SELECT IFNULL(City, 'Unknown') AS
CityName FROM Employees;
```

City	CityName
delhi	delhi
mumbai	mumbai
pune	pune
NULL	Unknown

Use IFNULL() (MySQL) or ISNULL() / COALESCE() in other databases.

QUICK REFERENCE

UPPER()	Convert text to uppercase
LOWER()	Convert text to lowercase
LENGTH()	Get length of string
ROUND()	Round numeric values
NOW()	Get current date & time
CONCAT()	Join strings
IFNULL()	Replace NULL with value

WHERE CAN FUNCTIONS BE USED?

- In SELECT clause
- In WHERE conditions
- In ORDER BY / GROUP BY
- In HAVING clause
- Inside other functions

PRO TIP

Functions help you write cleaner, shorter and more efficient SQL queries.

EXAMPLE – USING FUNCTIONS TOGETHER

```
SELECT UPPER(Name) AS Name,
       CONCAT(City, ' - ', IFNULL(ROUND(Salary, 0), 0)) AS Info,
       LENGTH(Name) AS NameLength,
       NOW() AS QueryTime
FROM Employees;
```

BENEFITS SUMMARY

- Reduces code length
- Improves readability
- Handles common tasks easily
- Makes queries powerful
- Saves time and effort

COMMON MISTAKES

- Using wrong function name
- Wrong number/type of arguments
- Ignoring NULL values
- Not checking return data type

SUPPORTED IN POPULAR DATABASES

Function	MySQL	SQL Server	Oracle	PostgreSQL
UPPER()	✔	✔	✔	✔
LOWER()	✔	✔	✔	✔
LENGTH()	✔	✔	✔	✔
ROUND()	✔	✔	✔	✔
NOW()	✔	✔	(SYSDATE)	✔
CONCAT()	✔	✔	✔	✔
IFNULL()	✔	ISNULL()	NVL()	COALESCE()

SUMMARY

SQL functions are powerful built-in tools that help you manipulate data easily and efficiently. Learn them, practice them and use them to write smart SQL queries!

LEARN • PRACTICE • USE FUNCTIONS • WRITE BETTER QUERIES

MASTER SQL FUNCTIONS – SIMPLIFY DATA, AMPLIFY RESULTS!

SQL TRIGGERS

Automate Actions on Database Events

A Trigger is a special type of stored procedure that automatically executes (or fires) when a specific event occurs in a table.

KEY CONCEPTS

TRIGGER
A trigger is a set of SQL statements that automatically runs when an event occurs on a table.

EVENTS
Triggers are fired by events like INSERT, UPDATE or DELETE on a table.

AUTOMATION
Helps automate tasks, maintain data integrity and enforce business rules.

DATABASE
Triggers are stored in the database and associated with a specific table.

ACCURACY
Ensures consistent and accurate data by executing actions automatically.

PERFORMANCE
Use wisely! Triggers can impact performance if overused or written inefficiently.

SAMPLE TABLE: Employees

EmpID	Name	Department	Salary
1	Alice	IT	60000
2	Bob	HR	50000
3	Charlie	Finance	55000

WHAT IS A TRIGGER?

A trigger is a database object that automatically executes a set of statements when a specific event occurs on a table.

WHY USE TRIGGERS?

- Enforce business rules
- Maintain audit logs
- Validate data automatically
- Keep data consistent
- Automate repetitive tasks

1. BEFORE INSERT TRIGGER

Executes before a new record is inserted into the table.

Event: BEFORE INSERT

```
CREATE TRIGGER trg_before_insert
BEFORE INSERT ON Employees
FOR EACH ROW
BEGIN
   SET NEW.JoinDate = NOW();
END;
```

How it works:
Before a new employee is inserted, the trigger sets the JoinDate to the current date & time.

Result Example

EmpID	Name	JoinDate
4	David	2025-05-25 14:30:00

2. AFTER INSERT TRIGGER

Executes after a new record is inserted into the table.

Event: AFTER INSERT

```
CREATE TRIGGER trg_after_insert
AFTER INSERT ON Employees
FOR EACH ROW
BEGIN
  INSERT INTO EmployeeLog
     (EmpID, Action, ActionTime)
  VALUES (NEW.EmpID, 'INSERT', NOW());
END;
```

How it works:
After a new employee is inserted, the trigger logs the action in EmployeeLog table.

EmployeeLog Example

LogID	EmpID	Action	ActionTime
1	4	INSERT	2025-05-25 14:30:00

3. AFTER UPDATE TRIGGER

Executes after an existing record is updated.

Event: AFTER UPDATE

```
CREATE TRIGGER trg_after_update
AFTER INSERT ON Employees
FOR EACH ROW
BEGIN
  INSERT INTO EmployeeLog
     (EmpID, Action, ActionTime)
  VALUES (NEW.EmpID, 'INSERT', NOW());
END;
```

How it works:
After an employee record is updated, the trigger logs the update action.

EmployeeLog Example

LogID	EmpID	Action	ActionTime
2	2	INSERT	2025-05-25 14:30:00

4. AFTER DELETE TRIGGER

Executes after a record is deleted from the table.

Event: AFTER DELETE

```
CREATE TRIGGER trg_after_delete
AFTER DELETE ON Employees
FOR EACH ROW
BEGIN
  INSERT INTO EmployeeLog
     (EmpID, Action, ActionTime)
  VALUES (OLD.EmpID, 'DELETE', NOW());
END;
```

How it works:
After an employee record is deleted, the trigger logs the delete action.

EmployeeLog Example

LogID	EmpID	Action	ActionTime
3	3	DELETE	2025-05-25 15:30:00

TRIGGER SYNTAX

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
  -- SQL statements
END;
```

TRIGGER EVENTS

Event	Description
BEFORE INSERT	Runs before a new row is inserted
AFTER INSERT	Runs after a new row is inserted
BEFORE UPDATE	Runs before a row is updated
AFTER UPDATE	Runs after a row is updated
BEFORE DELETE	Runs before a row is deleted
AFTER DELETE	Runs after a row is deleted

TRIGGER FLOW

DML Operation (INSERT, UPDATE, DELETE) → Trigger (Event Occurs) → Trigger Executes (Actions Performed) → Data Affected (Result Updated)

EXAMPLE: AUDIT LOG SCENARIO

We maintain an EmployeeLog table to track all changes.

EmployeeLog Table

LogID	EmpID	Action	ActionTime
1	4	INSERT	14:30:00
2	2	UPDATE	15:00:00
3	3	DELETE	15:30:00

Actions Tracked

- Who (EmpID)
- What Action (INSERT/UPDATE/DELETE)
- When (ActionTime)

BEST PRACTICES

- Keep triggers simple and focused.
- Avoid complex logic in triggers.
- Test triggers thoroughly.
- Use triggers for data integrity and auditing.
- Monitor trigger performance regularly.

COMMON USE CASES

- Audit logging
- Enforce business rules
- Update related tables
- Data validation
- Maintain history of changes

TIP: Triggers are powerful but should be used carefully to avoid unexpected side effects and performance issues.

Remember: With great power comes great responsibility!

Use Triggers Wisely!

TRANSACTIONS

ENSURE DATA INTEGRITY & RELIABILITY

A **Transaction** is a sequence of one or more SQL operations treated as a single unit of work. It either completes successfully (**COMMIT**) or is completely undone (**ROLLBACK**).

KEY CONCEPTS

TRANSACTION
A set of SQL operations performed as a single logical unit of work.

ATOMIC
All operations succeed or none. No partial changes.

CONSISTENCY
Maintains database rules and constraints before and after.

RELIABILITY
Ensures dependable results even in case of failures.

ISOLATION
Transactions don't interfere with each other.

DURABILITY
Committed changes are permanent, even after system failure.

EVENTS
BEGIN, COMMIT, ROLLBACK, SAVEPOINT control the flow.

SAMPLE TABLE: Accounts

AccountID	AccountName	Balance
1	Alice	10000
2	Bob	8000
3	Charlie	5000

WHAT IS A TRANSACTION?

A transaction groups multiple SQL statements into a single unit. It ensures that either all statements are executed successfully or none are applied to the database.

TRANSACTION COMMANDS

1. BEGIN

Starts a new transaction.

```
BEGIN;
```

Transaction Started

2. COMMIT

Saves all the changes permanently.

```
COMMIT;
```

Changes Saved

3. ROLLBACK

Undoes all changes made in the transaction.

```
ROLLBACK;
```

Changes Undone

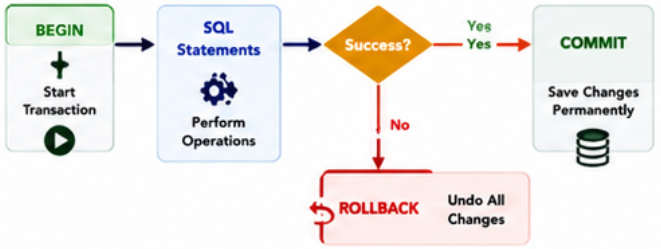
4. SAVEPOINT

Sets a point within a transaction to which you can roll back.

```
SAVEPOINT sp1;
```

Savepoint Created

TRANSACTION FLOW



ACID PROPERTIES

A — ATOMICITY
All or nothing. Either the entire transaction is completed or none.

C — CONSISTENCY
Database remains valid according to all defined rules and constraints.

I — ISOLATION
Transactions are isolated from each other.

D — DURABILITY
Once committed, changes are permanent.

EXAMPLE: TRANSFER MONEY FROM ALICE TO BOB

SCENARIO

Transfer ₹2000 from Alice (AccountID = 1) to Bob (AccountID = 2).

Alice → ₹2000 → Bob

If any step fails, no money should be transferred.

SUCCESS CASE (COMMIT)

```
BEGIN;
UPDATE Accounts SET Balance = Balance - 2000 WHERE AccountID = 1;
UPDATE Accounts SET Balance = Balance + 2000 WHERE AccountID = 2;
COMMIT;
```

Result

AccountID	AccountName	Balance
1	Alice	8000
2	Bob	10000

Changes Committed Successfully

FAILURE CASE (ROLLBACK)

```
BEGIN;
UPDATE Accounts SET Balance = Balance - 2000 WHERE AccountID = 1;
-- Some error occurs here (e.g., connection lost)
ROLLBACK;
```

Result

AccountID	AccountName	Balance
1	Alice	10000
2	Bob	8000

All Changes Rolled Back

SAVEPOINT EXAMPLE

```
BEGIN;
UPDATE Accounts  SET Balance = Balance - 1000
      WHERE AccountID = 1;
SAVEPOINT sp1;
UPDATE Accounts  SET Balance = Balance - 1500
      WHERE AccountID = 1;
-- Opps! Something went wrong
ROLLBACK TO sp1;    -- Go back to savepoint
COMMIT;     -- Commit up to savepoint
```

Use SAVEPOINT when you want partial rollback, not the entire transaction.

COMMON USE CASES

- Banking transactions (fund transfer)
- E-commerce orders & payments
- Inventory updates
- Data migration
- Any critical multi-step operation

BEST PRACTICES

- Keep transactions short and simple.
- Always handle exceptions and rollback.
- Use meaningful savepoint names.
- Commit only when all operations succeed.
- Test transactions thoroughly.

TRANSACTION ISOLATION LEVELS (OVERVIEW)

Level	Description
READ UNCOMMITTED	Allows dirty reads (lowest isolation)
READ COMMITTED	Prevents dirty reads
REPEATABLE READ	Prevents dirty & non-repeatable reads
SERIALIZABLE	Highest isolation, prevents all issues

QUICK REFERENCE

BEGIN	Start a new transaction
COMMIT	Save all changes
ROLLBACK	Undo all changes
SAVEPOINT name	Set a savepoint
ROLLBACK TO name	Rollback to a savepoint
RELEASE SAVEPOINT name	Remove a savepoint

KEY TAKEAWAY

Transactions guarantee Atomicity, Consistency and Reliability of your database operations.

PRO TIP: Use transactions whenever data integrity is critical. Always COMMIT on success, ROLLBACK on failure.

Without transactions, your data is always at risk!

COMMIT & ROLLBACK

Control Your Transactions, Protect Your Data

Transactions help keep your database safe and reliable. Use **COMMIT** to save changes permanently or **ROLLBACK** to undo changes if something goes wrong.

KEY CONCEPTS

COMMIT	ROLLBACK	SAVEPOINT	UNDO	CONFIRM	TRANSACTION	RECOVERY	DATABASE
Confirms and saves all changes permanently.	Undoes all changes made in the current transaction.	Sets a point within a transaction to which you can rollback.	Reverts changes made after the last SAVEPOINT.	Final confirmation that makes changes permanent.	A sequence of SQL operations treated as a single unit.	Helps restore the database to a consistent state.	Ensures data integrity, consistency and reliability.

WHAT IS A TRANSACTION?

A transaction is a group of SQL statements executed as a single unit of work. Either all succeed (COMMIT) or all fail (ROLLBACK).

ACID PROPERTIES

- **A** — **ATOMICITY**: All or nothing.
- **C** — **CONSISTENCY**: Database remains valid.
- **I** — **ISOLATION**: Transactions are isolated.
- **D** — **DURABILITY**: Committed changes are permanent.

1. COMMIT (CONFIRM CHANGES)

Saves all changes made in the current transaction permanently to the database.

Syntax

```
COMMIT;
```

Example

```
-- Start Transaction
BEGIN;
UPDATE Accounts SET Balance = Balance - 1000
  WHERE AccountID = 1;
UPDATE Accounts SET Balance = Balance + 1000
  WHERE AccountID = 2;
COMMIT;   -- Changes saved permanently
```

Result

AccountID	AccountName	Balance (Before)	Balance (After)
1	Alice	10000	9000
2	Bob	8000	9000

All changes are permanently saved.

2. ROLLBACK (UNDO CHANGES)

Undoes all changes made in the current transaction and restores the database to the previous state.

Syntax

```
ROLLBACK;
```

Example

```
-- Start Transaction
BEGIN;
UPDATE Accounts SET Balance = Balance - 1000
  WHERE AccountID = 1;
UPDATE Accounts SET Balance = Balance + 1000
  WHERE AccountID = 2;
ROLLBACK;  -- All changes undone
```

Result

AccountID	AccountName	Balance (Before)	Balance (After)
1	Alice	10000	10000
2	Bob	8000	8000

All changes are undone (rolled back).

3. SAVEPOINT (PARTIAL ROLLBACK)

Sets a point within a transaction. You can rollback to that point without affecting the whole transaction.

Syntax

```
SAVEPOINT savepoint_name;
ROLLBACK TO savepoint_name;
RELEASE SAVEPOINT savepoint_name;
```

Example

```
BEGIN;
UPDATE Accounts SET Balance = Balance - 500 WHERE AccountID = 1;
SAVEPOINT sp1;
UPDATE Accounts SET Balance = Balance - 300 WHERE AccountID = 1;
ROLLBACK TO sp1; -- Undo last update (300)
COMMIT;     -- First update (500) is saved
```

Result

AccountID	AccountName	Balance (Before)	Balance (After)
1	Alice	10000	9500

Rolled back to savepoint sp1, then committed.

TRANSACTION FLOW

WHEN TO USE

- When data integrity is critical
- When multiple related operations must succeed or fail together
- When you need to undo changes in case of errors
- During money transfers, bookings, order processing, etc.

COMMIT vs ROLLBACK vs SAVEPOINT

Feature	COMMIT	ROLLBACK	SAVEPOINT
Purpose	Save changes	Undo all changes	Set a point to rollback
Scope	Whole transaction	Whole transaction	Partial transaction
After Execution	Changes permanent	All changes undone	Can rollback to point
Use When	Everything is correct	Something went wrong	Need partial undo

COMMON USE CASES

- Banking Transactions (Fund Transfer, Withdrawals)
- Booking Systems (Flights, Hotels, Tickets)
- E-commerce Orders (Order Placement, Payments)
- Payroll Processing
- Inventory Management (Stock Updates)
- Any Critical Business Operations

BEST PRACTICES

- Always BEGIN a transaction.
- COMMIT only when you are sure.
- Use ROLLBACK on errors.
- Use SAVEPOINT for complex steps.
- Handle exceptions and rollback properly.

QUICK REFERENCE

```
BEGIN;                      -- Start Transaction
COMMIT;                     -- Save Changes
ROLLBACK;                   -- Undo All Changes
SAVEPOINT sp_name;          -- Set Savepoint
ROLLBACK TO sp_name;        -- Rollback to Savepoint
RELEASE SAVEPOINT sp_name;  -- Release Savepoint
```

PRO TIP: Always test your transactions. A well-managed transaction ensures data integrity and saves you from big problems!

Safe Data. Strong Database. Smart Transactions.

Without COMMIT, changes are temporary.
Without ROLLBACK, errors become permanent!

ACID PROPERTIES

The Foundation of Reliable Transactions

ACID properties ensure that database transactions are processed reliably and safely, even in case of errors, system failures, or concurrent operations.

KEY BENEFITS

- **Reliable Data** — Ensures accurate and trustworthy information.
- **Integrity** — Maintains correctness and consistency of the database.
- **Recovery** — Recovers from failures without losing important data.
- **Concurrency** — Allows multiple users to work without conflicts.
- **Database Safety** — Protects data from errors, crashes and system issues.

WHAT IS A TRANSACTION?

A transaction is a sequence of one or more SQL operations treated as a single unit of work.

It either completes successfully (COMMIT) or is completely undone (ROLLBACK).

WHY ACID MATTERS?

- Ensures data reliability
- Prevents data corruption
- Supports safe concurrency
- Protects against failures
- Builds user trust in data

EXAMPLE: BANK TRANSFER

ACID ensures the transfer is completed fully or not at all.

A ATOMICITY

All operations in a transaction are treated as a single unit. Either all succeed or all fail.

Example

If money is deducted from Account A but not added to Account B due to an error, the entire transaction is rolled back.

If Violated?

Partial updates occur, leading to incorrect data.

C CONSISTENCY

A transaction brings the database from one valid state to another valid state.

Example

Balance cannot be negative. All rules, constraints and relationships are maintained before and after the transaction.

If Violated?

Data rules break, causing inconsistent database.

I ISOLATION

Concurrent transactions do not interfere with each other. Each transaction is isolated.

Example

Two users transferring money at the same time will not see each other's intermediate changes.

If Violated?

Conflicting operations may lead to wrong results.

D DURABILITY

Once a transaction is committed, its changes are permanent, even if the system fails.

Example

After COMMIT, data is saved to disk. Even after a crash or power failure, data remains safe.

If Violated?

Committed data may be lost after system failure.

ACID IN ACTION

BEGIN TRANSACTION
↓
Perform Operations (INSERT, UPDATE, DELETE)
↓
COMMIT (All changes saved)
OR
↓
ROLLBACK (Undo all changes)

ACID = RELIABLE DATA

ACID properties work together to ensure reliable transactions and strong database integrity.

ACID SUMMARY TABLE

Property	Focus	Key Guarantee	Benefits
Atomicity (A)	All or Nothing	Either all operations succeed or none	Prevents partial updates
Consistency (C)	Data Rules	Database remains in a valid state	Maintains data integrity
Isolation (I)	Concurrency	Transactions do not interfere	Ensures accurate results
Durability (D)	Persistence	Committed data is permanent	Survives failures

ACID IN REAL LIFE

Banking	Ensures safe money transfers and balance accuracy.
E-Commerce	Ensures orders are placed correctly without conflicts.
Flight Booking	Ensures seat booking is confirmed once and safely.
Healthcare	Ensures patient data is accurate and never lost.

WHAT HAPPENS IF ACID IS NOT FOLLOWED?

Without Atomicity	Partial data saves cause big problems.
Without Consistency	Data becomes wrong and unreliable.
Without Isolation	Users get conflicting or incorrect results.
Without Durability	Data may be lost forever after failure.

PRO TIP: Always use transactions for important operations like payments, bookings, and data updates.

Good Transactions = Happy Users
ACID ensures trust in your database.

Remember: ACID is the backbone of every reliable database system.

Strong Database. Safe Data.
Better Trust. Always ACID!

NORMALIZATION

Design Better Databases

Normalization is the process of organizing data in a database to **reduce redundancy** and improve **data integrity**.

WHY NORMALIZE?

- **Reduce Redundancy** — Eliminates duplicate data and saves storage.
- **Improve Data Integrity** — Ensures accurate and consistent data.
- **Easier Maintenance** — Changes in data need to be made in one place only.
- **Better Performance** — Well-structured tables improve query efficiency.
- **Data Consistency** — Prevents anomalies and maintains consistency.
- **Reliable Design** — Builds a strong foundation for scalable systems.

WHAT IS NORMALIZATION?

Normalization is a step-by-step approach to organize data into tables to minimize redundancy and dependency.

It helps to ensure data integrity and easier data management.

KEY BENEFITS

- Reduces data redundancy
- Improves data integrity
- Avoids update anomalies
- Saves storage space
- Makes database easy to maintain

COMMON ANOMALIES (SOLVED BY NORMALIZATION)

- **Insert Anomaly** — Cannot insert data without other data.
- **Update Anomaly** — Same data exists in multiple rows.
- **Delete Anomaly** — Deleting data causes loss of information.

1NF (FIRST NORMAL FORM)

Eliminate repeating groups. Ensure each field has atomic (single) values.

Example (UNNORMALIZED)

Orders (OrderID, CustomerName, Products, Quantity)

OrderID	CustomerName	Products	Quantity
101	Ali	Pen, Pencil	2, 3
102	Sara	Book, Bag	1, 2

In 1NF

OrderID	CustomerName	Product	Quantity
101	Ali	Pen	2
101	Ali	Pencil	3
102	Sara	Book	1
102	Sara	Bag	2

All values are atomic. No repeating groups.

2NF (SECOND NORMAL FORM)

Be in 1NF and eliminate partial dependency. Every non-key field must be fully dependent on the whole primary key.

Example (1NF Table)

OrderDetails (OrderID, ProductID, ProductName, Quantity, Price)

OrderID	ProductID	ProductName	Quantity	Price
101	P1	Pen	2	10
101	P2	Pencil	3	5
102	P3	Book	1	50
102	P4	Bag	2	30

In 2NF (Split into two tables)

Orders

OrderID	CustomerName
101	Ali
102	Sara

OrderDetails

OrderID	ProductID	Quantity	Price
101	P1	2	10
101	P2	3	5
102	P3	1	50
102	P4	2	30

No partial dependency on composite key.

3NF (THIRD NORMAL FORM)

Be in 2NF and eliminate transitive dependency. Non-key fields should not depend on other non-key fields.

Example (2NF Table with Issue)

OrderDetails (OrderID, ProductID, ProductName, Quantity, Price, Category)

OrderID	ProductID	ProductName	Quantity	Price	Category
101	P1	Pen	2	10	Stationery
101	P2	Pencil	3	5	Stationery
102	P3	Book	1	50	Books
102	P4	Bag	2	30	Accessories

In 3NF (Remove transitive dependency)

OrderDetails

OrderID	ProductID	Quantity
101	P1	2
101	P2	3
102	P3	1
102	P4	2

Products

ProductID	ProductName	Price	Category
P1	Pen	10	Stationery
P2	Pencil	5	Stationery
P3	Book	50	Books
P4	Bag	30	Accessories

No transitive dependency. Data is more reliable.

BCNF (BOYCE-CODD NORMAL FORM)

For every dependency X → Y, X must be a super key. Stronger form of 3NF.

Example (3NF Table with Issue)

Enrollments (StudentID, CourseID, InstructorID, InstructorName)

StudentID	CourseID	InstructorID	InstructorName
S1	C1	I1	John
S1	C2	I2	Mary
S2	C1	I1	John

In BCNF (Decompose further)

StudentCourse

StudentID	CourseID
S1	C1
S1	C2
S2	C1

CourseInstructor

CourseID	InstructorID	InstructorName
C1	I1	John
C2	I2	Mary

Every determinant is a candidate key.

NORMALIZATION LEVELS SUMMARY

Normal Form	Focus	Key Rule
1NF	Atomicity	No repeating groups, atomic values
2NF	Partial Dependency	No partial dependency on part of key
3NF	Transitive Dependency	No transitive dependency on non-key fields
BCNF	Stronger Integrity	Every determinant is a candidate key

QUICK COMPARISON

Feature	1NF	2NF	3NF	BCNF
Repeating Groups Removed	✓	✓	✓	✓
Partial Dependency Removed	✗	✓	✓	✓
Transitive Dependency Removed	✗	✗	✓	✓
Stronger Integrity (Every Determinant is a Key)	✗	✗	✗	✓

BEST PRACTICES

- Start from 1NF and move step by step.
- Normalize till 3NF for most applications.
- Use BCNF for complex and large databases.
- Balance between normalization and performance.
- Always test with real data scenarios.

REMEMBER

Normalization is not about creating more tables, it's about creating the RIGHT tables.

- Better Design
- Better Integrity
- Better Reliability

GOOD DESIGN TODAY, RELIABLE DATA TOMORROW! | Normalize to Optimize | Integrity is Priority | Smart Design, Strong System | Reliable Data, Strong Business

DENORMALIZATION

Optimize for Read Performance

Denormalization is the process of intentionally adding redundant data or grouping data to improve read performance and simplify queries, at the cost of increased storage and potential data redundancy.

WHY DENORMALIZE?

- **Better Performance** — Faster queries and response time.
- **Fast Queries** — Complex joins are reduced or eliminated.
- **Improved Read Speed** — Data is pre-combined for quick access.
- **Optimization** — Optimize for read-heavy workloads.
- **Simpler Queries** — Reports and dashboards become easier and faster.
- **Scalability for Reads** — Handle more read requests efficiently.
- **Business Advantage** — Better user experience and quicker decisions.

WHAT IS DENORMALIZATION?

Denormalization adds redundant data or combines tables to reduce joins and speed up read operations.

It is useful in read-heavy systems like reporting, dashboards, and analytics.

PROS & CONS

PROS

- Faster read performance
- Simpler and fewer joins
- Easy reporting and analytics
- Better for large datasets
- Improves user experience

CONS

- Duplicate data increases storage
- Data inconsistency risk
- More effort on updates/deletes
- Complex write operations
- Maintenance can be harder

WHEN TO DENORMALIZE?

- Read operations are much more frequent than write operations.
- Complex joins are slowing down queries.
- Reports, dashboards, or aggregations need to be fast.
- Data is mostly static or changes rarely.

NORMALIZED DESIGN (More Joins, Slower Reads)

Orders

OrderID	CustomerID	OrderDate
101	1	2024-05-01
102	2	2024-05-02
103	1	2024-05-03

Customers

CustomerID	CustomerName	City
1	Ali	Delhi
2	Sara	Mumbai
3	John	Pune

OrderItems

OrderID	ProductID	Quantity
101	10	2
101	20	1
102	10	1
103	30	3

Query (Requires Multiple Joins)

```
SELECT o.OrderID, c.CustomerName, o.OrderDate, p.ProductName, oi.Quantity
FROM Orders o
JOIN Customers c ON o.CustomerID = c.CustomerID
JOIN OrderItems oi ON o.OrderID = oi.OrderID
JOIN Products p ON oi.ProductID = p.ProductID;
```

Slower Queries — More Joins

DENORMALIZED DESIGN (Fewer Joins, Faster Reads)

Orders_Denorm

OrderID	OrderDate	CustomerID	CustomerName	City	ProductID	ProductName	Quantity
101	2024-05-01	1	Ali	Delhi	10	Pen	2
101	2024-05-01	1	Ali	Delhi	20	Book	1
102	2024-05-02	2	Sara	Mumbai	10	Pen	1
103	2024-05-03	1	Ali	Delhi	30	Bag	3

Query (Single Table, Faster)

```
SELECT OrderID, CustomerName, OrderDate, ProductName, Quantity
FROM Orders_Denorm
WHERE OrderDate BETWEEN '2024-05-01' AND '2024-05-31';
```

Faster Queries — Single Table

KEY CONCEPTS

- **Duplicate Data** — Some data is intentionally repeated.
- **Pre-Joined Data** — Data from multiple tables is stored together.
- **Aggregated Data** — Summaries or totals are stored for quick access.
- **Materialized Views** — Precomputed results are stored for performance.
- **Purpose** — Optimize read performance at the cost of redundancy.

DENORMALIZATION TECHNIQUES

- **Add Redundant Columns** — Add frequently used related data in same table.
- **Merge Tables** — Combine related tables to reduce joins.
- **Summary Tables** — Store pre-calculated aggregates for fast reporting.
- **Materialized Views** — Store the result of complex queries for reuse.
- **Lookup Tables** — Duplicate small reference data locally.

COMMON USE CASES

- Reporting & Analytics
- Dashboards & BI Tools
- E-commerce Product Views
- Data Warehousing
- Search & Filtering Systems
- Real-time Read Applications

EXAMPLE: E-COMMERCE

Normalized (More Joins)

Orders
Customers
Products
OrderItems

→

Denormalized (Faster Reads)

Orders_Denorm
(Contains customer, product & order details together)

THINGS TO REMEMBER

- Use denormalization when reads >> writes.
- Always ensure data consistency.
- Use proper processes or triggers for updates.
- More redundancy = More maintenance.
- Balance between performance and storage.

NORMALIZATION vs DENORMALIZATION

Aspect	Normalization	Denormalization
Goal	Reduce redundancy	Improve read performance
Tables	More tables, related	Fewer tables, combined
Joins	More joins required	Fewer or no joins
Performance	Slower reads, faster writes	Faster reads, slower writes
Storage	Less storage	More storage
Complexity	Complex queries	Simple queries
Best For	Transactional systems	Reporting & analytics

QUICK SUMMARY

- Denormalization improves performance by reducing joins.
- It increases redundancy but makes queries faster.
- Ideal for read-heavy systems like reporting and dashboards.
- Plan carefully and maintain data consistency.
- Use only where performance gains are critical.

BEST PRACTICES

- Denormalize only after analyzing query performance.
- Keep write operations in normalized tables if possible.
- Use ETL or batch jobs to sync denormalized data.
- Document all redundant data and dependencies.
- Monitor storage and consistency regularly.

REMEMBER

Normalization is for accuracy and integrity. Denormalization is for speed and performance. Choose the right design for the right workload!

Optimize for Reads. | Use Denormalization Wisely. | Fast Queries, Happy Users. | Balance Performance & Integrity. | Smart Design, Strong Database!

BACKUP & RESTORE

PROTECT YOUR DATA, ENSURE RECOVERY

Backup is creating a copy of your database for safety.
Restore is bringing back that data when needed.
Backup Today, Secure Tomorrow! ✅

KEY BENEFITS

- **Data Protection** — Protects against data loss due to crashes, errors or corruption.
- **Quick Recovery** — Restore your database quickly and get back to normal.
- **Business Continuity** — Keeps your business running even during major failures.
- **Peace of Mind** — Regular backups give you confidence and security.
- **Data Integrity** — Ensures your data remains accurate and consistent.
- **Security** — Backups can be encrypted and stored securely.
- **Compliance** — Helps meet legal and compliance requirements.

BACKUP

Create a copy of your database for safekeeping.

RESTORE

Restore your database from a backup file.

MYSQLDUMP (CLI TOOL)

Command-line utility used to backup databases.

EXPORT

Export data or database into an SQL file.

IMPORT

Import SQL file into a database.

RECOVERY

Recover lost or corrupted data using backup.

SECURITY

Secure your backups and protect sensitive data.

BACKUP USING MYSQLDUMP

```
# Backup entire database
mysqldump -u username -p database_name > backup.sql

# Backup with options (structure + data)
mysqldump -u username -p database_name --routines
--triggers --events > full_backup.sql

# Backup all databases
mysqldump -u username -p --all-databases > all_db_backup.sql
```

RESTORE / IMPORT USING MYSQL

```
# Login to MySQL
mysql -u username -p

# Restore database
mysql> CREATE DATABASE database_name;
mysql> USE database_name;
mysql> SOURCE backup.sql;   -- or
mysql -u username -p database_name < backup.sql
```

EXPORT (SELECTED DATA)

```
# Export specific table data
mysqldump -u username -p database_name table_name > table.sql

# Export with WHERE condition
mysqldump -u username -p database_name table_name
--where="id > 100" > filtered_data.sql
```

IMPORT DATA (TABLE)

```
# Import specific table data
mysql -u username -p database_name < table.sql
```

BACKUP TYPES

Type	Description
Full Backup	Backup entire database (structure + data)
Schema Backup	Backup only structure (tables, views, etc.)
Data Backup	Backup only data
Incremental Backup	Backup only changes since last backup
Binary Log Backup	Backup binary logs for point-in-time recovery

COMMON MYSQLDUMP OPTIONS

Option	Descriptions
--databases db1 db2	Backup specific databases
--tables table1 table2	Backup specific tables
--no-data	Backup structure only
--no-create-info	Exclude CREATE statements
--add-drop-table	Add DROP TABLE before CREATE TABLE
--routines --triggers --events	Include stored routines, triggers, events

BACKUP BEST PRACTICES

- ✅ Take regular backups (daily/weekly).
- ✅ Test your backups by restoring on test system.
- ✅ Store backups in multiple locations.
- ✅ Use meaningful file names with date & time.
- ✅ Automate backups using scripts or schedulers.
- ✅ Monitor backup size and free disk space.

SECURE YOUR BACKUPS

- Store backups in secure, access-controlled locations.
- Encrypt backup files using tools like GPG.
- Do not store backups on the same server.
- Regularly verify backup integrity.
- Limit access to backup files and folders.

SAMPLE BACKUP WORKFLOW

Create Backup (mysqldump) → Store Safely (Local/Cloud) → Verify Backup (Integrity Check) → Schedule Regular Backups → Ready for Recovery

QUICK COMMAND REFERENCE

Backup DB:
```
mysqldump -u user -p db_name > backup.sql
```

Restore DB:
```
mysql -u user -p db_name < backup.sql
```

Backup All DBs:
```
mysqldump -u user -p --all-databases > all_backup.sql
```

RECOVERY SCENARIOS

- **Accidental Deletion** — Restore from latest backup.
- **Server Crash** — Restore database and restart service.
- **Corrupted Data** — Restore from last known good backup.
- **Data Migration** — Backup → Transfer → Restore on new server.

REMEMBER

A good backup strategy today can save you from big problems tomorrow!

Backup Regularly, Restore Confidently!

PRO TIP: A backup is worth a thousand fixes. Make it regular, keep it safe, and test it always!

Backup Today | Recover Anytime | Stay Secure | Keep Your Data Safe!

USER MANAGEMENT

Manage Users. Control Access. Secure Your Database.

User management allows you to create, modify and remove database users and control their access privileges securely.

Login · Password · Privileges · Security · Authentication

WHY USER MANAGEMENT IS IMPORTANT?

- **Security** — Restrict access to authorized users only.
- **Control Access** — Grant specific privileges to users as needed.
- **Data Protection** — Protect sensitive data from unauthorized access.
- **Audit & Tracking** — Track user activities and maintain accountability.
- **Flexibility** — Easily manage users, roles and passwords anytime.
- **Better Performance** — Assign proper access to improve system performance.

WHAT IS A DATABASE USER?

A database user is an account that allows you to connect to the database and perform operations based on the privileges granted.

USER COMPONENTS

- **Username** — Unique identity of the user.
- **Host** — Where the user can connect from.
- **Password** — Used for authentication.
- **Privileges** — Permissions on databases/tables.
- **Authentication Plugin** — Method used for authentication.

1. CREATE USER

Creates a new database user with a password.

```
CREATE USER 'username'@'host'
 IDENTIFIED BY 'password';
```

Example:

```
CREATE USER 'raj'@'localhost' IDENTIFIED BY 'Raj@123';
CREATE USER 'dev'@'%' IDENTIFIED BY 'Dev@123';
```

Hosts

'localhost'	User can connect only from the local server.
'%'	User can connect from any host.
'192.168.1.%'	User can connect from IP range.

2. ALTER USER

Modifies an existing user (change password or options).

```
ALTER USER 'username'@'host'
 IDENTIFIED BY 'new_password';
```

Example:

```
ALTER USER 'raj'@'localhost' IDENTIFIED BY 'NewRaj@123';
ALTER USER 'dev'@'%' IDENTIFIED BY 'NewDev@123';
```

Other Alter Options

Rename User	`RENAME USER 'old_user'@'host' TO 'new_user'@'host';`
Change Host	`ALTER USER 'raj'@'localhost' RENAME TO 'raj'@'%';`
Change Authentication Plugin	`ALTER USER 'raj'@'%' IDENTIFIED WITH mysql_native_password BY 'pass';`

3. DROP USER

Removes a user from the system.

```
DROP USER 'username'@'host';
```

Example:

```
DROP USER 'raj'@'localhost';
DROP USER 'dev'@'%';
```

Important

- Dropping a user removes all their access.
- Make sure no application depends on the user before dropping.

4. PASSWORD MANAGEMENT

Change or reset user password.

```
SET PASSWORD FOR 'username'@'host'
  = 'new_password';
```

Example:

```
SET PASSWORD FOR 'raj'@'localhost' = 'Raj@1234';
SET PASSWORD FOR 'dev'@'%' = 'Dev@1234';
```

Check Password Policy

```
SHOW VARIABLES LIKE 'validate_password%';
```

AUTHENTICATION PLUGINS

- mysql_native_password (Default in older versions)
- caching_sha2_password (Default in MySQL 8.0+)
- sha256_password
- auth_socket (Linux only)

LOGIN EXAMPLE

```
mysql -u username -p
-h host -P port
```

Prompts for password.

SECURITY TIPS

- Use strong passwords.
- Grant only required privileges.
- Use specific hosts instead of '%'.
- Regularly review and drop unused users.
- Enable password expiry policies.

CHECK USERS

```
SELECT User, Host, plugin
FROM mysql.user;
```

Or

```
SHOW GRANTS FOR 'username'@'host';
```

COMMON USER COMMANDS

Command	Description
CREATE USER	Creates a new user.
ALTER USER	Modifies user properties.
DROP USER	Deletes a user.
RENAME USER	Renames a user.
SET PASSWORD	Changes user password.
SHOW GRANTS	Shows privileges of a user.

USER PRIVILEGES (EXAMPLES)

Privilege	Description
SELECT	Allows to read data.
INSERT	Allows to insert data.
UPDATE	Allows to update data.
DELETE	Allows to delete data.
ALL PRIVILEGES	Grants all privileges.
GRANT OPTION	Allows to grant privileges to others.

GRANT PRIVILEGES TO USER

```
GRANT privilege ON database.table
TO 'username'@'host';
```

Example:

```
GRANT SELECT, INSERT, UPDATE ON mydb.*
TO 'raj'@'localhost';
FLUSH PRIVILEGES;
```

Always run `FLUSH PRIVILEGES;` after changes to apply immediately.

REVOKE PRIVILEGES FROM USER

```
REVOKE privilege ON database.table
FROM 'username'@'host';
```

Example:

```
REVOKE DELETE ON mydb.*
FROM 'raj'@'localhost';
FLUSH PRIVILEGES;
```

USER MANAGEMENT WORKFLOW

Create User → Set Password → Grant Privileges → User Access DB → Review & Maintain

KEY TAKEAWAY

Good user management = Strong security. Create users carefully, grant limited access, and review regularly.

- **Strong Passwords** — Use complex and unique passwords.
- **Least Privilege** — Give only the access that is necessary.
- **Regular Review** — Check users and privileges periodically.
- **Secure Authentication** — Use modern authentication plugins.
- **Backup & Recovery** — Always backup before making major user changes.

PERMISSIONS

Control Access. Protect Data.

Permissions (Privileges) define what actions a user is allowed to perform on database objects.
Use **GRANT** to give access and **REVOKE** to remove it.

KEY CONCEPTS

Security
Restrict access to authorized users only.

Privileges
Specific rights to perform actions on database objects.

Access Control
Grant only the necessary access (Least Privilege).

Data Protection
Protect data from unauthorized access or changes.

Roles
Group of privileges that can be assigned to users.

Accountability
Track and manage who can do what in the database.

COMMON PRIVILEGES

SELECT (Read)	View data from tables.
INSERT (Write)	Insert new records.
UPDATE (Write)	Modify existing records.
DELETE (Write)	Delete records.
REFERENCES	Create foreign keys.
EXECUTE	Execute stored procedures.
ALL PRIVILEGES	All available privileges.

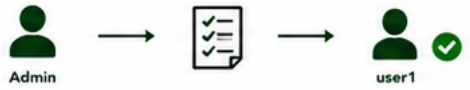
GRANT – GIVE PERMISSIONS

Syntax:

```
GRANT privilege_list ON object TO user [WITH GRANT OPTION];
```

Example:

```
GRANT SELECT, INSERT, UPDATE ON sales.* TO user1@'localhost';
```



WITH GRANT OPTION
Allows the user to grant the same privileges to other users.

```
GRANT SELECT ON employees TO user1@'localhost' WITH GRANT OPTION;
```

REVOKE – REMOVE PERMISSIONS

Syntax:

```
REVOKE privilege_list ON object FROM user;
```

Example:

```
REVOKE INSERT, UPDATE ON sales.* FROM user1@'localhost';
```

Important
Revoking a privilege does not remove it from other users.

PRIVILEGES ON OBJECTS

Privilege	Table	View	Database	Function	Routine
SELECT	✓	✓	✓ (SHOW)	✓	✓
INSERT	✓	✗	✗	✗	✗
UPDATE	✓	✗	✗	✗	✗
DELETE	✓	✗	✗	✗	✗
CREATE	✗	✗	✓	✓	✓
DROP	✗	✗	✓	✓	✓
ALL PRIVILEGES	✓	✓	✓	✓	✓

ROLES – GROUP OF PRIVILEGES

Create a role:

```
CREATE ROLE report_role;
```

Grant privileges to role:

```
GRANT SELECT ON sales.* TO report_role;
```

Assign role to user:

```
GRANT report_role TO user1@'localhost';
```

Revoke role from user:

```
REVOKE report_role FROM user1@'localhost';
```

PRIVILEGE LEVELS (SCOPE)

Table Level
Applies to specific tables.
`GRANT SELECT ON db.table TO user;`

Database Level
Applies to all tables in a database.
`GRANT SELECT ON db.* TO user;`

Server Level
Applies to the entire server.
`GRANT ALL PRIVILEGES ON *.* TO user;`

Column Level (MySQL 8.0+)
Applies to specific columns.
`GRANT SELECT (col1, col2) ON db.table TO user;`

EXAMPLES OF GRANT & REVOKE

Description	Command
Give read access on a table	`GRANT SELECT ON employees TO user1;`
Give write access on a table	`GRANT INSERT, UPDATE, DELETE ON employees TO user1;`
Give all privileges on a database	`GRANT ALL PRIVILEGES ON company.* TO user1;`
Allow user to grant others	`GRANT SELECT ON employees TO user1 WITH GRANT OPTION;`
Remove a privilege	`REVOKE DELETE ON employees FROM user1;`
Remove all privileges	`REVOKE ALL PRIVILEGES ON company.* FROM user1;`

USERS & THEIR PRIVILEGES

User	Host	Privileges	Granted On	With Grant Option
user1	localhost	SELECT, INSERT, UPDATE	sales.*	No
report_user	%	SELECT	reports.*	Yes
admin	localhost	ALL PRIVILEGES	*.*	Yes

Use `SHOW GRANTS FOR 'user'@'host';` to see user privileges.

SECURITY BEST PRACTICES

- Grant only necessary privileges.
- Use least privilege principle.
- Avoid using ALL PRIVILEGES.
- Use roles to simplify management.
- Review and revoke unused access.
- Audit user permissions regularly.

COMMON COMMANDS CHEAT SHEET

Action	Command
Show all users	`SELECT user, host FROM mysql.user;`
Show grants for a user	`SHOW GRANTS FOR 'user1'@'localhost';`
Create user	`CREATE USER 'user1'@'localhost' IDENTIFIED BY 'pass123';`
Drop user	`DROP USER 'user1'@'localhost';`

REMEMBER: Good permissions = Strong security
Give the right access to the right users!

Read (SELECT) | **Write** (INSERT, UPDATE, DELETE) | **Admin** (ALL PRIVILEGES) | **Roles** (Group Privileges) | **Security** (Protect Data)

PERFORMANCE OPTIMIZATION

Optimize Queries. Improve Speed. Scale Better.

Performance optimization improves the speed, efficiency and scalability of your database applications.

Better Performance = Better Experience! ✅

KEY OPTIMIZATION AREAS

Indexes	Query Optimization	Execution Plan	Caching	Joins	Avoid SELECT *	Speed & Performance
Speeds up data retrieval.	Write efficient queries.	Understand how queries run.	Store data for faster access.	Use joins wisely for better results.	Fetch only needed columns.	Better optimization = Faster applications.

1. INDEXES – SPEED UP SEARCHES

- Indexes reduce the time to find rows.
- Create indexes on columns used in WHERE, JOIN, ORDER BY, GROUP BY.
- But too many indexes can slow down INSERT, UPDATE, DELETE.

Example:

```
CREATE INDEX idx_emp_name
ON employees(name);
```

Best Practice
Index columns that are frequently searched, sorted or used in joins.

2. QUERY OPTIMIZATION – WRITE SMART QUERIES

- Write queries that are simple and efficient.
- Use proper filtering, avoid unnecessary subqueries.
- Use LIMIT to restrict rows.
- Avoid functions on indexed columns.

Poor Query:

```
SELECT * FROM orders WHERE YEAR(order_date) = 2024;
```

Better Query:

```
SELECT * FROM orders
WHERE order_date >= '2024-01-01' AND order_date < '2025-01-01';
```

Best Practice
Filter early, fetch less and use indexed columns.

3. EXECUTION PLAN – KNOW HOW IT RUNS

- Execution plan shows how the query is executed.
- Helps find full table scans, missing indexes, and extra operations.
- Use EXPLAIN to analyze queries.

Example:

```
EXPLAIN SELECT * FROM employees
WHERE department_id = 10;
```

Check For	What It Means
type = ALL	Full table scan (slow)
Using index	Using index (good)
Using filesort	Extra sorting (optimize)
Using temporary	Extra temp table (optimize)

4. CACHING – REDUCE LOAD & IMPROVE SPEED

- Caching stores frequently used data in memory.
- Reduces database load and improves response time.
- Use query cache, application cache (Redis, Memcached).

Example:

- MySQL Query Cache (Deprecated in 8.0)
- Redis Cache
- Application-Level Cache

Best Practice
Cache data that is read frequently but changes rarely.

5. JOINS – USE WISELY

- Joins are powerful but can be slow with large tables.
- Use INNER JOIN instead of subqueries when possible.
- Ensure joined columns are indexed.

Example:

```
SELECT e.name, d.dept_name
FROM employees e
INNER JOIN departments d
ON e.department_id = d.id;
```

Best Practice
Index columns used in JOIN conditions.

6. AVOID SELECT * – FETCH ONLY WHAT YOU NEED

- SELECT * fetches all columns (unnecessary data).
- It uses more bandwidth, memory and CPU.
- Always select only required columns.

Avoid:

```
SELECT * FROM employees;
```

Better:

```
SELECT id, name, email FROM employees;
```

Best Practice
Fetch only required columns, especially in large tables.

7. OTHER OPTIMIZATION TIPS

- Use LIMIT to restrict results.
- Use proper data types.
- Avoid OR in WHERE (use UNION instead).
- Denormalize when necessary (with care).
- Keep statistics updated (ANALYZE TABLE).
- Optimize database configuration (Buffer size, Cache size, etc).
- Regularly clean and archive old data.

COMMON SLOW QUERY REASONS

Issue	Solution
Missing Indexes	Add proper indexes
Full Table Scan	Add WHERE + Index
Too Many Joins	Optimize joins / Use subqueries
SELECT *	Select only needed columns
Large Result Set	Use LIMIT / Pagination
Outdated Statistics	Run ANALYZE TABLE
Unoptimized Queries	Rewrite and test queries

QUERY OPTIMIZATION WORKFLOW

Write Query → Check Execution Plan (EXPLAIN) → Find Bottlenecks → Optimize (Query / Index / Structure) → Test & Monitor Performance

SAMPLE – IMPROVED QUERY

Slow Query (Unoptimized)

```
SELECT * FROM orders o
JOIN customers c ON o.customer_id =c.id
WHERE YEAR(o.order_date) = 2024
ORDER BY o.order_date DESC;
```

Optimized Query

```
SELECT o.id, o.amount, o.order_date, c.name
FROM orders o
JOIN customers c ON o.customer_id =c.id
WHERE o.order_date >= '2024-01-01'
AND o.order_date < '2025-01-01'
ORDER BY o.order_date DESC
LIMIT 100;
```

What Improved?

- Used date range (index friendly)
- Selected only needed columns
- Added LIMIT
- Better performance!

PERFORMANCE CHECKLIST

- Proper indexes created
- Queries are optimized
- Execution plan analyzed
- Unnecessary columns removed
- Joins are optimized
- Caching implemented
- Database & server settings tuned
- Regular monitoring & testing

TOOLS FOR OPTIMIZATION

- EXPLAIN (Execution Plan)
- SHOW PROFILE (Query Analysis)
- Slow Query Log
- Performance Schema
- MySQL Workbench
- Third Party Tools (Percona Monitoring, New Relic, etc.)

REMEMBER

Fast queries + Smart indexes + Proper caching = High performance!

Optimize Today, Scale Tomorrow! ✅

Optimize Queries | Improve Speed | Reduce Load | Better Performance | Happy Users | Smart Queries, Strong Databases!

COMMON SQL ERRORS

Find. Fix. Learn. Write Better Queries.

Errors are normal, but understanding them makes you a better SQL developer!

✅ Detect ✅ Understand ✅ Fix ✅ Optimize ✅

WHY ERRORS HAPPEN?

- **Syntax Mistakes** — Typos and wrong command structure.
- **Data Issues** — NULL values, duplicates or invalid data.
- **Constraint Violations** — Breaking rules like keys and relationships.
- **Connection Problems** — Network, server or configuration issues.
- **Timeout & Performance** — Long running queries or locked resources.
- **Permission Issues** — Not enough rights to run queries.
- **Logic & Debugging** — Incorrect logic or unhandled conditions.

1 SYNTAX ERROR

Occurs when SQL is written in the wrong format.

Example

```
SELEC * FROM users;
```

```
Error: You have an error in your SQL syntax near 'SELEC'
```

How to Fix

- Check spelling of SQL keywords.
- Ensure proper commas, brackets, quotes and semoiolons.

Prevention Tip

Use an SQL formatter and validate queries before running.

2 NULL ERROR

Happens when NULL value is not handled properly.

Example

```
SELECT name FROM users
WHERE age > 18;
```

```
Error: Unknown column 'age' in 'where clause'
```

How to Fix

- Check for NULL values using IS NULL.
- Use IFNULL(), COALESCE() or proper conditions.

Prevention Tip

Always handle NULL values in queries and conditions.

3 DUPLICATE KEY ERROR

Occurs when inserting a duplicate value in UNIQUE or PRIMARY KEY.

Example

```
INSERT INTO users(id, email)
VALUES(1, 'jeet@example.com');
```

```
Error: Duplicate entry '1' for key 'PRIMARY'
```

How to Fix

- Ensure unique values.
- Use INSERT IGNORE or ON DUPLICATE KEY UPDATE if needed.

Prevention Tip

Validate unique data before insert or update operations.

4 FOREIGN KEY ERROR

Occurs when foreign key value does not exist in parent table.

Example

```
INSERT INTO orders(user_id)
VALUES(999);
```

```
Error: Cannot add or update a child row: a foreign key constraint fails
```

How to Fix

- Insert valid parent record first.
- Check foreign key relationships.

Prevention Tip

Validate referential integrity in all relationships.

5 CONNECTION ERROR

Occurs when unable to connect to the database server.

Example

```
mysql -u root -p
```

```
Error: Can't connect to MySQL server on 'localhost' (10061)
```

How to Fix

- Check server status.
- Verify host, port, username and password.

Prevention Tip

Ensure database service is running and credentials are correct.

6 TIMEOUT ERROR

Happens when a query takes too long to execute.

Example

```
SELECT * FROM huge_table;
```

```
Error: Query execution was interrupted (timeout)
```

How to Fix

- Optimize query.
- Add indexes.
- Increase timeout limit if needed.

Prevention Tip

Optimize queries and use indexes for better performance.

7 PERMISSION ERROR

Occurs when user does not have enough privileges.

Example

```
DROP TABLE users;
```

```
Error: You don't have permission to drop table 'users'
```

How to Fix

- Grant required permissions.
- Login with correct user role.

Prevention Tip

Use GRANT statement carefully and follow least privilege rule.

8 DEBUGGING TIPS

Use these techniques to find and fix errors faster.

- ✅ Read error message carefully.
- ✅ Check query step by step.
- ✅ Use LIMIT to test small data.
- ✅ Use EXPLAIN to analyze query.
- ✅ Print intermediate results.
- ✅ Use IDE or SQL tools.
- ✅ Log and track errors.

ERROR QUICK REFERENCE

Error Type	Error Code (MySQL)	Meaning	Impact	Fix Priority
Syntax Error	1064	Wrong SQL syntax	Query fails	High
NULL Error	1054	Invalid or missing value	Wrong result / Fails	High
Duplicate Key	1062	Duplicate entry in unique key	Insert/Update fails	High
Foreign Key Error	1452	Violation of foreign key	Insert/Update fails	High
Connection Error	2003 / 10061	Cannot connect to server	No access	High
Timeout Error	3024	Query took too long	Query aborted	Medium
Permission Error	1142	Access denied for user	Operation blocked	High

HOW TO DEBUG SQL ERRORS?

1. **Read Error Message** — Understand what went wrong.
2. **Check Query Step by Step** — Run small parts of the query.
3. **Verify Data & Structure** — Check tables, columns, keys and data.
4. **Fix & Test Again** — Make changes and run again.
5. **Optimize & Prevent** — Improve query and avoid future errors.

USEFUL SQL SNIPPETS FOR TROUBLESHOOTING

Check Table Structure

```
DESCRIBE table_name;
```

Check Indexes

```
SHOW INDEX FROM table_name;
```

Check Foreign Keys

```
SHOW CREATE TABLE table_name;
```

Check Server Status

```
SHOW STATUS;
```

Kill Long Running Query

```
KILL query_id;
```

See All Privileges

```
SHOW GRANTS FOR 'user'@'host';
```

BEST PRACTICES TO AVOID ERRORS

- ✅ Always validate queries before running on large data.
- ✅ Use meaningful names for tables and columns.
- ✅ Follow proper database normalization.
- ✅ Use transactions to maintain data integrity.
- ✅ Backup database regularly.
- ✅ Monitor performance and optimize queries.
- ✅ Keep database and tools updated.

Clean Queries
Strong Database
Happy Developer! 🙂

Remember: Every error is a learning opportunity.

Debug today, Better tomorrow!

Write Clean SQL Think Before You Run

Secure Data Ensure Integrity

Optimize Performance Deliver Speed

SQL INTERVIEW QUESTIONS

Most Asked SQL Questions in Interviews

Practice these important questions and boost your SQL skills!

WHY THESE TOPICS?

- **Core Concepts** — Build strong fundamentals
- **Database Design** — Design efficient and normalized databases
- **Performance** — Write optimized and faster queries
- **Data Integrity** — Maintain accurate and reliable data
- **Interview Ready** — Frequently asked in real interviews
- **Career Growth** — Essential for DBAs, Developers & Analysts

1. JOINS

Q1. What is a JOIN in SQL?
Ans: JOIN is used to combine rows from two or more tables based on a related column between them.

Q2. What are the types of JOINs?
Ans: INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN, CROSS JOIN.

Example (INNER JOIN)

```
SELECT e.name, d.dept_name
FROM employees e
INNER JOIN departments d
ON e.dept_id = d.dept_id;
```

2. INDEXES

Q1. What is an Index?
Ans: An index is a database object that improves the speed of data retrieval operations on a table.

Q2. What are the types of Indexes?
Ans: PRIMARY INDEX, UNIQUE INDEX, CLUSTERED INDEX, NON-CLUSTERED INDEX.

Example

```
CREATE INDEX idx_emp_name
ON employees(name);
```

3. NORMALIZATION

Q1. What is Normalization?
Ans: Normalization is the process of organizing data to reduce redundancy and improve data integrity.

Q2. What are the Normal Forms?
Ans: 1NF, 2NF, 3NF, BCNF, 4NF, 5NF.

Goal

- ✔ Eliminate redundancy
- ✔ Improve data integrity
- ✔ Ensure data consistency

4. PRIMARY KEY

Q1. What is a Primary Key?
Ans: A primary key is a column (or combination of columns) that uniquely identifies each row in a table.

Q2. Can a Primary Key be NULL?
Ans: No, Primary Key cannot contain NULL values.

Example

```
CREATE TABLE employees (
  emp_id INT PRIMARY KEY,
  name VARCHAR(50)
);
```

5. FOREIGN KEY

Q1. What is a Foreign Key?
Ans: A foreign key is a column (or set of columns) in one table that refers to the primary key in another table.

Q2. Why use Foreign Key?
Ans: It maintains referential integrity between related tables.

Example

```
ALTER TABLE employees
ADD CONSTRAINT fk_dept
FOREIGN KEY (dept_id)
REFERENCES departments(dept_id);
```

6. VIEWS

Q1. What is a View in SQL?
Ans: A view is a virtual table based on the result of a SQL query.

Q2. Why use Views?
Ans: To simplify complex queries, restrict data access, and improve security.

Example

```
CREATE VIEW emp_view AS
SELECT emp_id, name, dept_id
FROM employees;
```

7. TRIGGERS

Q1. What is a Trigger?
Ans: A trigger is a stored procedure that automatically executes (fires) when an event occurs in a table.

Q2. Types of Triggers?
Ans: BEFORE trigger, AFTER trigger, INSTEAD OF trigger.

Example (AFTER INSERT)

```
CREATE TRIGGER trg_after_insert
AFTER INSERT ON employees
FOR EACH ROW
INSERT INTO emp_log VALUES (NEW.emp_id, NOW());
```

8. TRANSACTIONS

Q1. What is a Transaction?
Ans: A transaction is a sequence of SQL operations performed as a single logical unit of work.

Q2. What are the properties?
Ans: ACID – Atomicity, Consistency, Isolation, Durability.

Example

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
COMMIT; -- or ROLLBACK;
```

QUICK REFERENCE

Topic	Key Point
Joins	Combine rows from multiple tables
Indexes	Improve search performance
Normalization	Reduce redundancy, improve integrity
Primary Key	Uniquely identifies each row
Foreign Key	Links tables and enforces relationships
Views	Virtual tables for simplified access
Triggers	Automate actions on events
Transactions	Ensure reliable and consistent operations

COMMON INTERVIEW TIPS

- ✔ Understand the concept, not just the syntax.
- ✔ Practice writing queries on real datasets.
- ✔ Know when and why to use each feature.
- ✔ Explain with examples.
- ✔ Discuss performance and best practices.

SAMPLE INTERVIEW QUESTION

1. Write a query to find duplicate records in a table.
2. How would you find the second highest salary?
3. Explain the difference between DELETE, TRUNCATE and DROP.
4. How does a LEFT JOIN differ from a LEFT OUTER JOIN?
5. Write a query to get departments with more than 5 employees.
6. How do you optimize a slow running query?

BEST PRACTICES

- ★ Always use indexes on search/filter columns.
- ★ Avoid SELECT * in production queries.
- ★ Normalize your database design.
- ★ Use transactions for data consistency.
- ★ Test and optimize your queries regularly.

Be Curious. Keep Practicing. | Write Better Queries. | Build Strong Databases. | Optimize Performance. | Crack Your Next Interview!

SQL BEST PRACTICES

Write Better SQL. Build Better Databases.

Follow these best practices to improve performance, security, and maintainability of your database.

WHY FOLLOW BEST PRACTICES?

- **Better Performance** — Faster queries and optimized resources.
- **High Security** — Protect data from threats and errors.
- **Data Integrity** — Ensure accurate and consistent data.
- **Easy Maintenance** — Clean code is easy to manage.
- **Scalability** — Handle growth with confidence.
- **Professionalism** — Write clean SQL like a pro!

1. MEANINGFUL NAMES

- Use clear and descriptive names for tables, columns, and aliases.
- Avoid short or confusing names like t1, a, x.
- Use consistent naming conventions.

Good:

```
SELECT employee_name, hire_date
FROM employees;
```

2. INDEXES

- Create indexes on frequently searched columns.
- Avoid over-indexing.
- Regularly review and optimize indexes.

Example:

```
CREATE INDEX idx_emp_name
ON employees (employee_name);
```

3. PARAMETERIZED QUERIES

- Use parameterized queries to prevent SQL injection.
- Never concatenate user input directly in queries.
- Use prepared statements.

Example (Prepared Statement):

```
SELECT * FROM users WHERE id = ?;
```

4. AVOID SELECT *

- Select only the columns you need.
- Reduces unnecessary data transfer and improves performance.
- Makes queries more readable.

Avoid:

```
SELECT * FROM orders;
```

Use:

```
SELECT order_id, order_date, total FROM orders;
```

5. BACKUP REGULARLY

- Always keep regular backups of your database.
- Test restore process to ensure backup is valid.
- Store backups securely.

Example (MySQL):

```
mysqldump -u user -p database_name > backup.sql
```

6. NORMALIZATION

- Normalize databases to reduce redundancy.
- Use proper relationships between tables.
- Improves data integrity and saves storage.

Example Normal Forms:

1NF → 2NF → 3NF → BCNF

7. SECURITY

- Use strong passwords and roles.
- Grant only necessary privileges.
- Validate and sanitize all inputs.
- Keep software and DB updated.

Example:

```
GRANT SELECT, INSERT ON db.*
TO 'app_user'@'localhost';
```

8. DOCUMENTATION

- Document database schema, tables, and relationships.
- Comment your SQL code for clarity.
- Keep documentation up to date.

Example (Comment):

```
-- Get all active employees hired after 2023
SELECT * FROM employees WHERE status = 'active'
AND hire_date > '2023-01-01';
```

QUICK SUMMARY

Use meaningful names | Index wisely | Use parameterized queries | Avoid SELECT * | Backup regularly | Normalize your data | Keep your data secure | Document everything

GOLDEN RULES

- Think before you write.
- Test your queries.
- Optimize continuously.
- Security is everyone's responsibility.
- Keep it simple.
- Readability > Complexity.
- Performance matters!

Remember:
Good SQL makes your application **FAST, SECURE and RELIABLE.**

Clean Code | Fast Queries | Secure Data | Strong Databases | Better Results

SQL DEVELOPER TOOLS

Powerful Tools for Database Developers

Manage databases, write queries, design schemas, and improve productivity with these top SQL tools.

WHY USE THESE TOOLS?

- **Boost Productivity** — Work faster with smart features.
- **Database Management** — Easily manage, monitor and maintain databases.
- **Query Development** — Write, test and optimize SQL queries.
- **Performance Tuning** — Analyze and improve query performance.
- **Secure & Reliable** — Work with security and accuracy.
- **Multi-Platform** — Supports multiple databases and OS.

1 MySQL WORKBENCH

MySQL Workbench

- Official tool for MySQL.
- Visual database design.
- SQL development & execution.
- Server administration.
- Performance monitoring.

Best For: MySQL Database Design, Development & Administration

2 phpMyAdmin

phpMyAdmin

- Web-based interface.
- Manage MySQL/MariaDB.
- Import/Export databases.
- User & privilege management.
- Easy to use & open source.

Best For: Web Developers & Quick Database Management

3 DBeaver

DBeaver

- Universal database tool.
- Supports many databases.
- Advanced SQL editor.
- Data export & import.
- ER diagrams & connection manager.

Best For: Developers working with Multiple Databases

4 pgAdmin

pgAdmin

- Official tool for PostgreSQL.
- Query tool & schema designer.
- Database monitoring.
- Backup & restore.
- Extensive admin features.

Best For: PostgreSQL Administration & Development

5 SQL Server Management Studio (SSMS)

Microsoft SQL Server

- Official tool for SQL Server.
- Query editor with IntelliSense.
- Database design & diagrams.
- Performance tuning.
- Backup, restore & deployment.

Best For: SQL Server Development & Administration

6 Oracle SQL Developer

ORACLE SQL Developer

- Official tool for Oracle DB.
- Full-featured SQL IDE.
- PL/SQL development.
- Data modeling & reports.
- Performance & debugging.

Best For: Oracle Detabase Developers & Administrators

7 SQLite Browser

SQLite Browser

- Lightweight & simple.
- Open-source SQLite manager.
- Browse & edit data easily.
- Execute SQL queries.
- Import/Export databases.

Best For: SQLite Database Management

8 Visual Studio Code (VS Code)

VS Code

- Lightweight code editor.
- SQL extensions support.
- Connect to databases.
- Run & debug queries.
- Great for developers.

Best For: Developers who want a Security & Customizable IDE

TOOLS COMPARISON

	Tool	Database Support	Platform	Open Source	Best Use
1	MySQL Workbench	MySQL	Windows, macOS, Linux	✓	Design, Development, Admin
2	phpMyAdmin	MySQL, MariaDB	Web (Any OS)	✓	Quick Web-based Management
3	DBeaver	Multiple (50+ DBs)	Windows, macOS, Linux	✓	Multi-DB Development
4	pgAdmin	PostgreSQL	Windows, macOS, Linux	✓	PostgreSQL Admin & Dev
5	SQL Server (SSMS)	Microsoft SQL Server	Windows	✗	SQL Server Admin & Dev
6	Oracle SQL Developer	Oracle Database	Windows, macOS, Linux	✓	Oracle DB Development
7	SQLite Browser	SQLite	Windows, macOS, Linux	✓	SQLite Management
8	VS Code	Multiple (via Extensions)	Windows, macOS, Linux	✓	Coding & Query Editor

HOW TO CHOOSE THE RIGHT TOOL?

- If you work mostly with MySQL → Use MySQL Workbench or phpMyAdmin
- For PostgreSQL databases → Use pgAdmin
- For Microsoft SQL Server → Use SSSMS
- For Oracle databases → Use Oracle SQL Developer
- For multiple databases → Use DBeaver
- For modern, lightweight coding → Use VS Code

QUICK TIPS

- Always backup your database before making changes.
- Use parameterized queries to prevent SQL injection.
- Analyze query performance regularly.
- Keep your tools and database servers updated.
- Use version control for SQL scripts.

Right Tool + Best Practices = Better Performance, Security & Productivity!

Choose the right tool. Build better databases.

Write Better Queries. Optimize Performance.

Secure Your Data. Protect Your Work.

Manage Databases Like a Pro.

Better Tools. Better Developers.

SQL QUERY EXECUTION FLOW

How a SQL Query Runs from Start to Finish

From writing a query to getting the final result!

WHY UNDERSTAND THIS FLOW?

- Better Query Writing
- Performance Optimization
- Easy Troubleshooting
- Efficient Resource Use
- Faster Results

1 WRITE QUERY

User writes a SQL query and sends it to the database.

Example:

```
SELECT name, salary
FROM employees
WHERE dept_id = 10;
```

2 PARSER

Parser checks the syntax and validates the query.

Checks for:

- Syntax errors
- Table & column existence
- Permissions

3 OPTIMIZER

Optimizer analyzes possible execution paths and chooses the best one.

Considers:

- Indexes
- Statistics
- Cost estimation
- Join order

4 EXECUTION PLAN

Optimizer creates an execution plan – a step-by-step strategy to execute the query.

Shows:

- Access methods
- Join operations
- Sorts & filters
- Estimated cost

5 DATABASE ENGINE

Execution plan is sent to the database engine for execution.

Engine:

- Allocates resources
- Executes operations
- Manages memory
- Handles transactions

6 FETCH DATA

Engine reads the required data from tables, indexes, or other sources.

Data Access via:

- Index Scan
- Table Scan
- Joins
- Temporary storage

7 RESULT SET

Matching rows are processed and prepared as the result set.

Operations:

- Filtering
- Sorting
- Aggregation
- Grouping

8 OUTPUT

Final result is returned to the user in the output format.

Displayed as:

- Table
- JSON
- CSV
- Application Output

EXAMPLE FLOW

1. User: `SELECT * FROM orders WHERE amount > 1000;`
2. Parser: Validates syntax and checks objects.
3. Optimizer: Chooses the best plan using indexes on amount.
4. Plan: Index Scan on amount > 1000 → Fetch rows.
5. Engine: Executes plan using allocated memory & CPU.
6. Fetch Data: Reads matching rows from index/table.
7. Result Set: Rows are filtered, sorted (if needed).
8. Output: Result returned to user/application.

INTERNAL PROCESS (BEHIND THE SCENES)

CPU → Memory → Disk I/O → Locking & Concurrency → Logging & Recovery

- CPU Executes Operations
- Memory Management
- Disk I/O Operations
- Locking & Concurrency
- Logging & Recovery

The optimizer's goal is to get the same result with the least cost (time, CPU, I/O, memory).

KEY POINTS

- Good indexes = Faster execution.
- Up-to-date statistics = Better plans.
- Avoid SELECT * = Less data, faster results.
- Understand execution plan for optimization.
- Test queries with different conditions.

Simple Query In, Powerful Result Out!

Write Smart Queries | **Use Indexes Wisely** | **Optimize Regularly** | **Secure & Reliable** | **Better Performance, Better Applications!**

SQL DATABASE ARCHITECTURE

How SQL Databases Work Behind the Scenes

From Client Request to Results – The Complete Flow

WHY THIS ARCHITECTURE MATTERS?

- Efficient Query Processing
- Data Security & Integrity
- High Performance & Scalability
- Reliable Data Storage
- Easy Management & Maintenance

1 CLIENT

- Applications
- Web / Mobile App
- Users

SQL Query

Results Returned

2 SQL SERVER

- **Connection Manager** – Handles connections from clients
- **Query Parser** – Checks syntax and validates SQL queries
- **Query Optimizer** – Creates the best execution plan
- **Execution Engine** – Executes the plan and processes data
- **Result Manager** – Formats and returns results to client

Optimized Query Plan

Data Results

3 DATABASE

TABLES

Data is organized in tables (rows & columns)

ID	Name	Age
1	Alice	25
2	Bob	30
3	Carol	28
...	...	...

INDEXES

Indexes improve search speed and performance

Types:
- Clustered Index
- Non-Clustered Index
- Unique Index
- Full-Text Index

QUERIES

SQL queries interact with the data

Examples:
- SELECT
- INSERT
- UPDATE
- DELETE
- JOIN

RELATIONSHIPS

Relations defined using Primary & Foreign Keys

Orders ↕ Customers

1 ----- N

Read / Write Data

Data Fetched

4 STORAGE ENGINE

- **Data Storage** – Stores actual data on disk
- **Transaction Manager** – Ensures ACID properties
- **Lock Manager** – Manages concurrency and data locks
- **Buffer Manager** – Caches data in memory for speed
- **Recovery Manager** – Handles backup, recovery & logging

ARCHITECTURE FLOW SUMMARY

1. Client sends SQL query to SQL Server.
2. SQL Server parses, optimizes and creates execution plan.
3. Execution engine accesses the database (tables, indexes).
4. Storage engine reads/writes data from disk and manages it.
5. Data is processed and results are generated.
6. Results are formatted by Result Manager.
7. Results are returned back to the client.

KEY COMPONENTS

Component	Purpose
Client	Sends queries and receives results
SQL Server	Parses, optimizes and executes queries
Database	Stores data in tables, indexes, and metadata
Storage Engine	Handles data storage, retrieval & transactions
Indexes	Speeds up data retrieval operations
Queries	Interact with data using SQL commands
Results	Output returned to the client/application

BENEFITS OF THIS ARCHITECTURE

- Fast and optimized query execution
- Efficient data storage and retrieval
- High concurrency and data consistency
- Scalable for small to enterprise systems
- Secure and reliable data management

PRO TIP

Well-designed architecture = Better Performance, Scalability, Security & Reliability.

- Design Smart – Normalize Data
- Index Wisely – Speed Up Queries
- Secure Data – Use Roles & Permissions
- Monitor & Optimize – Keep Performance High
- Backup Regularly – Ensure Data Safety

SQL DEVELOPMENT WORKFLOW

From Idea to Production – Step by Step

A complete workflow for building and managing SQL databases.

WHY FOLLOW A WORKFLOW?

- **Better Planning** — Structured approach reduces errors.
- **Higher Quality** — Tested and optimized queries.
- **Data Security** — Backup and security at every step.
- **Better Performance** — Optimized for speed and scalability.
- **Team Collaboration** — Easy to maintain and hand over.

1 DESIGN DATABASE

- Analyze requirements
- Identify entities and relationships
- Create ER Diagram
- Define data types and rules

Deliverable: ER Diagram & Database Design

2 CREATE TABLES

- Write CREATE TABLE statements
- Define Primary Keys and Constraints
- Set Foreign Keys and Indexes
- Review schema

Deliverable: Database Schema (Tables, Keys, Indexes)

3 INSERT DATA

- Insert initial data
- Use proper data validation
- Handle NULLs and defaults
- Verify inserted data

Deliverable: Populated Tables (Seed Data)

4 WRITE QUERIES

```
SELECT *
FROM Users
WHERE Age > 18;
```

- Write SELECT, INSERT, UPDATE, DELETE
- Use Joins, Subqueries, CTEs, Views
- Follow coding standards
- Review and peer check

Deliverable: SQL Queries & Scripts

5 TEST

- Test queries with different scenarios
- Validate results
- Check edge cases and errors
- Ensure adta integrity

Deliverable: Test Reports & Verified Results

6 OPTIMIZE

- Analyze query performance
- Use Indexes effectively
- Optimize Joins and Subqueries
- Remove redundant code

Deliverable: Optimized Queries & Indexes

7 BACKUP

- Create regular backups
- Test restore process
- Store backups securely
- Document backup strategy

Deliverable: Backup Files & Recovery Plan

8 DEPLOY

- Deploy to staging environment
- Run final tests
- Deploy to production
- Monitor and validate

Deliverable: Live Database (Production)

WORKFLOW OVERVIEW

Design Database → Create Tables → Insert Data → Write Queries → Test → Optimize → Backup → Deploy

Monitor, Maintain & Improve Continuously

BEST PRACTICES

- Plan before you code.
- Use meaningful names for database objects.
- Follow normalization principles.
- Write clean and efficient queries.
- Test thoroughly before deployment.
- Keep backups and document everything.
- Continuously monitor and optimize.

KEY TIPS

- Use Version Control (Git) for SQL scripts.
- Automate backups and deployments.
- Monitor performance in production.
- Review and refactor queries regularly.
- Keep learning and stay updated.

SUCCESS FORMULA

Good Design
+ Clean Code
+ Testing
+ Optimization
+ Security

= Reliable, Scalable SQL Solutions

PRO TIP
A good workflow today saves hours of debugging tomorrow!

- Plan Well — Avoid Rework
- Test Early — Find Issues Fast
- Optimize Always — Improve Performance
- Backup Regularly — Stay Safe
- Deploy Confidently — Deliver Value

SQL QUICK REFERENCE

★ Your Everyday Cheat Sheet for SQL

Fast Syntax Reference | Boost Productivity | Write Better Queries | For All SQL Developers

SQL BASICS

- SQL is not case sensitive (but values are).
- End most commands with a semicolon (;).
- Use meaningful names for tables/columns.
- Test your queries before using in production.

1 SELECT

Retrieve data from a table.

```
SELECT column1, column2
FROM table_name
WHERE condition;
-- All columns
SELECT * FROM table_name;
```

Example:

```
SELECT id, name, email
FROM employees
WHERE department = 'IT';
```

2 INSERT

Insert new data into a table.

```
INSERT INTO table_name
(column1, column2, ...)
VALUES (value1, value2, ...);
```

Example:

```
INSERT INTO employees
(name, email, department, salary)
VALUES ('Amit', 'amit@example.com',
'IT', 60000);
```

3 UPDATE

Update existing data.

```
UPDATE table_name
SET column1 = value1,
    column2 = value2
WHERE condition;
```

Example:

```
UPDATE employees
SET salary = 65000
WHERE id = 101;
```

4 DELETE

Delete data from a table.

```
DELETE FROM table_name
WHERE condition;
```

Example:

```
DELETE FROM employees
WHERE id = 101;
```

7 FUNCTIONS (Common)

```
COUNT()     -- Number of rows
SUM()       -- Total of values
AVG()       -- Average of values
MIN()       -- Minimum value
MAX()       -- Maximum value
UPPER()     -- Convert to upper case
LOWER()     -- Convert to lower case
LENGTH()    -- Length of string
NOW()       -- Current date & time
COALESCE()  -- Return first non-NULL value
```

5 JOINS

Combine rows from two or more tables.

INNER JOIN	Returns matching records from both tables.	`SELECT * FROM A INNER JOIN B ON A.id = B.a_id;`
LEFT JOIN	All records from left table and matches from right.	`SELECT * FROM A LEFT JOIN B ON A.id = B.a_id;`
RIGHT JOIN	All records from right table and matches from left.	`SELECT * FROM A RIGHT JOIN B ON A.id = B.a_id;`
FULL JOIN	All records when there is a match in either table.	`SELECT * FROM A FULL OUTER JOIN B ON A.id = B.a_id;`

6 GROUP BY

Group rows that have the same values into summary rows.

```
SELECT column, COUNT(*)
FROM table_name
GROUP BY column
HAVING condition
ORDER BY column;
```

Example:

```
SELECT department, COUNT(*) AS total_emp
FROM employees
GROUP BY department
HAVING COUNT(*) > 5
ORDER BY total_emp DESC;
```

8 TRANSACTIONS

Manage a group of operations as a single unit.

`BEGIN;`	Start a new transaction
`COMMIT;`	Save all changes
`ROLLBACK;`	Undo all changes
`SAVEPOINT sp1;`	Set a save point
`ROLLBACK TO sp1;`	Rollback to save point

WHERE Operators

=	<>	>	<	>=	<=	BETWEEN	LIKE	IN	IS NULL
Equal	Not Equal	Greater	Less	Greater or Equal	Less or Equal	Between a range	Pattern matching	Match any value list	Value is NULL

Other Useful Clauses

ORDER BY	Sort result set
ASC / DESC	Ascending / Descending order
DISTINCT	Return unique values
LIMIT	Limit number of rows returned
OFFSET	Skip rows (used with LIMIT)
AS	Alias (rename column or table)

Tip: Use meaningful aliases and proper indexing for better performance.

QUICK SUMMARY

Retrieve Data | Add Data | Modify Data | Remove Data | Combine Data | Summarize Data | Use Functions | Ensure Integrity

★ **Remember:** Good queries + Good design + Good indexes = Fast & Reliable Database!

Practice SQL Daily. Master the Basics. Build Amazing Applications!